ANSON THESIS

(dedicated to TN, Dijkstra)

PMK 14 Aug 75

Sent by: Ed Anson
Computer Graphics
Research Project
Katholieke Universiteit
Nijmegen
The Netherlands
Note thanks to TN, p.iii.

INTERACTIVE CREATION OF STRUCTURED PROGRAMS BY MEANS OF STEP-WISE REFINEMENT

by

Ed Anson

INTERACTIVE CREATION OF STRUCTURED PROGRAMS BY MEANS OF STEP-WISE REFINEMENT

by

Ed Anson

A THESIS

Presented to the Faculty of

The Graduate College in the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Department of Computer Science

Under the Supervision of Professor George Nagy

Lincoln, Nebraska

August, 1975

ABSTRACT

Natural-language communication is an attempt to map a richly structured concept into a linear medium such as speech or normal writing. The communication with machines via programming languages is a similar mapping. However, due to the improving economy of graphics displays and supporting computer hardware, a more richly structured medium is becoming feasible. Such a medium can support a structured presentation of programs as well as a structured methodology for their construction. A prototype system known as WHIMSI has been constructed and used to test one approach to structured program composition and presentation. WHIMSI is an acronym for WHolly Interactive Multiply Structured Information. The system uses an IBM 2250 graphics display terminal to support the activity of composing and modifying programs. Displays and controls deal with program segments as units, explicitly recognizing and manipulating their structure. This differs from previous programming aids which deal with the program's linear text structure instead. Preliminary tests demonstrate that the use of WHIMSI may at least double programmer efficiency in composing correct programs of a moderate size.

"Computers are extremely flexible and powerful tools and many feel that their application is changing the face of the earth. I would venture the opinion that as long as we regard them primarily as tools, we might grossly underestimate their significance. Their influence as tools might turn out to be but a ripple on the surface of our culture, whereas I expect them to have a much more profound influence in their capacity of intellectual challenge!"

Edsger W. Dijkstra
Notes on Structured Programming

"If computers are the wave of the future, displays are the surfboards."

Theodor H. Nelson
Dream Machines

☆

ACKNOWLEDGEMENTS

Great appreciation is due to the following people, without whose contributions this work would have been impossible.

Edsger W. Dijkstra

George Nagy

Theodor H. Nelson ★

Hari Sahasrabuddhe

CONTENTS

LIST OF FIGURES

INTRODUCTION

Programming is a complicated matter. It is sometimes even difficult and taxing to the intellect.

The principal objective of a programmer is to cause a process which produces a desired result. More specifically, the goal is a set of symbols which correctly specifies to a computer what is to be done. Considerable analysis may be required to discover what sort of process will give the effect desired.

Incidental to this aim (yet nonetheless essential) is the preparation of detailed specifications, flow charts, structure diagrams and other documents to explain the functioning of the program. Hopefully, good documentation will facilitate the verification and possible future modification of the process to be performed.

All this is further complicated by the fact that the actual specification to the computer must be in a binary code not easily written or understood by the (human) programmer. The process of translating program specifications into the appropriate machine code thus requires a good deal of precision and attention to detail.

Furthermore, modification of this code may require a great deal of labor. Each step in this process is subject to human error. It is thus only right that programmers get what help they can from the computer, which is characteristically precise and methodical and able to perform many error - free operations in a short time.

Such help is provided under the control of special programs (eg. assemblers, compilers, text editors) which may be referred to as programming aids.

The topic of this thesis is the nature and use of a new programming aid named WHIMSI. The name is an acronym for WHolly Interactive Multiply Structured Information.

The multiple structure implied by the system's name is basically a tree, augmented by arbitrary cross referencing or links. The tree represents the successive refinements of a structured program. That is, the program is defined by a _segment_ which is partitioned into steps or alternatives, or repeated until a criterion is met. Each segment includes an abstract description ("comment"), provided by the programmer, which specifies the action of that segment. Links provide a one-line description of the segment linked to, and provide a path to that segment.

Rather than represent a program text in the conventional sequential format, WHIMSI explicitly portrays its structure on a graphics display screen. As described in part three, each type of subdivision is represented by a distinct graphic symbol. Each successive subdivision of the process is in turn represented by the corresponding symbol. The result is a nested display representing the structure of a program.

Since the entire program cannot be effectively viewed at once, WHIMSI provides a means to move the view from point to point within the structure, and to control the depth of display. The viewpoint may be moved to any segment on display by simply pointing at it with the light pen. Thus any nested segment may be transformed easily into a full-screen view, from which the original display may be returned by use of a single command. A view of a segment not in the display may also be obtained by pointing at a link directed toward it.

WHIMSI allows the user to deal explicitly with a program's structure, creating and deleting entire segments with a single command, or rearranging them to alter the program's structure. A modified subdivision of a segment may be retained for reference, and reinstated if desired, allowing multiple versions of a program to be maintained in a simple manner.

Although an alphanumeric keyboard is available, it is used only to enter program text and comments. All commands to the WHIMSI system are entered by means of function keys and light pen selection. Several commands thus require only a single key stroke. More complicated commands require a series of operations, each of which is prompted by a query from the system.

<u>Overview.</u> In order to explicate the need for a system such as WHIMSI, and to help argue its appropriateness to the programming task, a brief historical review is given in part one. Some previous and current programming aids are examined, with special attention to interactive tools.

Building on this background, part two explains the basic concepts of WHIMSI and provides a more thorough rationale. Discussion in this part focuses on the services which a programming aid should provide, and on how they can be provided. Due to limitations of the available hardware and to the short time available for this project, several of the desirable features are not implemented. The restrictions of services in the present implementation are thus discussed in explaining the specific nature of WHIMSI.

Part three is a complete description of the use of WHIMSI. The basic structures are described, and the command language is explained. It thus serves as a user manual for the system.

The description of the program use is followed in part four by a sample dialog as recorded by WHIMSI during the construction of a program. Although the full flavor of the interactive dialog cannot be perceived from this static presentation, some indication can be obtained concerning its nature.

The conclusions following this sample relate to the evaluation of the present implementation of WHIMSI and to considerations for its modification and extension.

Four appendices are included for the benefit of those readers who may be interested in more detailed descriptions of the system. They serve in conjunction with the source listings as documentation of the WHIMSI system.

PART I:

HISTORICAL DEVELOPMENT

Many different tools have been created to assist programmers in their task of producing programs.

Some of these tools are conceptual and others are embodied in programs to perform specific chores for the programmer. Others, such as high - level programming languages, are both since they are generally implemented as compilers.

Purely intellectual tools (such as structured programming and structured design) are not considered in this part. Instead, the emphasis here is on programs created to help create programs. The conceptual tools are considered in part two.

Programming languages and their translators are considered here, along with program text editors (with and without extra functions) and some on-line editors of structured text. Some other tools are mentioned but not described in depth, as they are not yet well developed as practical tools.

Each of the tools described here is considered in terms of its associated _fantic space_ and the _fantic controls_ it provides.

These terms relate to _fantics_, "the art and science of presentation" (Nelson, 1973). Their root derives from the Greek word _phainein_, meaning to show, and from _phantazein_, to render visible or to present to the mind.

Fantic space is defined (Nelson, 1974) as "the space and relationships sensed by a viewer of any medium, or a user in any presenting or responding environment." *Fantic controls* (Nelson, 1973) are "any controls whose correspondence to the realm affected is restructured or mediated by fantic structure." *Fantic structure* is subsequently defined as "the structure of a presentation or presentational system, whether experienced by the user, intended by the designer, or discovered later on by somebody else, or an abstraction never suspected."

The steering wheel of an automobile is thus a fantic control, since it translates the circular motion of steering into the coordinated movements of the axles of two wheels. The fantic structure of the steering mechanism is simply a steering of the vehicle in the same direction as the turning of the wheel. And this structure is one element of the fantic space associated with the operation of an automobile.

The constructs employed by any programming aid may be analyzed in terms of the manner in which they mediate between the perceived fantic space and the actual process that results. They may be further evaluated in terms of the nature of the fantic space, its seeming naturalness and ease of use.

It is such an analysis which leads almost inevitably to the design of some system such as WHIMSI.

Translators and Programming Languages

There is considerable evidence that the fantic space presented by machine code is less than optimal for programmers.

A survey of 2000 programming errors made by professional programmers programming in machine code (reported by Weinberg, 1971, p. 204) showed that 65 percent were attributable to "housekeeping" errors such as miscoded operation codes, misplaced flags and incorrect addresses and offsets.

The simplest of translators, an assembler, relieves the programmer of responsibility for these details, greatly improving accuracy in the programs produced. More important though, is that the programmer no longer must think in terms of bits and op-codes and offsets; he may think instead in terms of the sequence of operations to be performed. Alternatively we may say that the fantic space for programming is shifted (by an assembler) closer in structure to that of the process to be created.

Despite the improvement afforded by an assembler, the programmer remains obliged to specify an entire process in terms of individual, minute operations at the machine level. In the case of simple problems (or where extreme efficiency is essential) this is acceptable. However, most programs of typical complexity are such that the programmer may not wish to deal with them so microscopically. This is especially true when some sequence of operations is coded many times with little or no variation.

Macro processors, when added to assemblers, tend to alleviate this short-coming of assembly languages. They allow the programmer virtually to create his own language for expressing the process to be performed. By creation of a hierarchy of macros referencing macros and finally generating machine instructions, a programmer may be able in some cases to adjust the fantic structure of his medium very nearly to the structure of the problem to be solved.

The degree to which this is true depends in part on the flexibility of the macro processor, and on the nature of the problem area. In any case, the problem remains of how to conveniently keep track of the large numbers of macros required and to recall the specific parameters required by each according to the results desired. A further problem occurs to a person attempting to read another's program. The set of macros constructed is virtually an entire language which must be mastered before the program can be read. Otherwise, constant cross referencing is required between sections of code and the related macros.

These factors, together with the tendency of most macro languages to be somewhat maladapted to certain important problem areas, led to the development of higher - level programming languages and compilers for them.

Programmers were somewhat skeptical (Weinberg, 1971, p. 51) of the first promises (circa 1956) that a program was about to be available which would allow a mathematical programmer to express his problem in mathematical formulas. But FORTRAN was perfected nonetheless. The programming of mathematical problems subsequently became easier since the fantic structure of a FORTRAN statement was much closer to the programmer's thinking than could be obtained from assemblers.

FORTRAN is still a standard among mathematical programmers, although it is being supplanted by APL. COBOL and various other specialized languages have subsequently been developed for business purposes, and now there are specialized languages for nearly every identifiable problem area. The motivation of each seems to be to allow the use of symbols which conform as closely as possible with the natural structure of thinking in each problem area.

This goal is well expressed by Zahn (1975): "It is useful to speak of the "conceptual distance" between program text and action sequences or between problem definition and program text. The programmer who wants some measure of confidence in the reliability of his program must bridge both these conceptual distances. It follows that a major goal of programming language design should be to help reduce both these distances."

The tendency in special - purpose language design is to reduce the degree to which the expression of the process is explicitly procedural. The "less procedural" languages allow the user/programmer to specify the problem, leaving to the translator/interpreter the job of specifying the appropriate procedure.

Languages such as these have been implemented (Leavenworth and Sammet, 1974) for business applications (MAPL), artificial intelligence (PLANNER and CONNIVER), simulation (SIMSCRIPT) and for set - oriented problem solving (SETL and MADCAP).

Floyd (1967) reports software which allows the specification of non-deterministic processes. At any given point, the action may result from a variable chosen randomly from a (usually small) finite set. Unlike procedural languages however, the consequences of each possible choice are explored. The first resulting chain of events which solves the problem is allowed to produce its output. Such a language is used for game playing and various combinatoric problems.

Such languages tend to bridge the conceptual gap between the problem and the process to solve it. Nevertheless, there are other considerations which render these languages less than ideal.

The foremost of these is that problems don't always fit neatly into the areas of competence for these languages.

Sometimes a problem requires application of more than one discipline to its solution. And there frequently arise new problems not anticipated in the design of any special - purpose language.

Due to this, general - purpose languages must often be used in order to solve new problems, and these languages often fail to be convenient. Consequently, software becomes increasingly expensive.

Volumes have been published relating to the problems of programming and programming languages. Many divers languages have been designed in an attempt to solve some of the problems. All suffer from the same intrinsic difficulty: They attempt to present a richly structured concept (or process) in a linear medium. Consequently, thus those parts of the program which are most closely related are not necessarily displayed close together, and the functional binding of program components may not be made clear.

The usefulness of this function is argued by Weinberg (1971, p. 224-225). His argument is that the utility of features in a language such as compactness, locality and linearity, are important to the readability of the program. He reports, for instance, that even the insertion of comments (which are essential to understanding) often tends to obscure the program to the reader by physically separating related information. This observation is borne out experimentally by Newsted.

Two aspects of the implicit structure of a process are not easily represented in a linear medium. These are alternation and repetition. Each requires (at best) two bracketing statements or symbols for each component. When a segment of a program becomes lengthy and complicated, containing several layers of nested elements, the appropriate grouping of operations is not easily perceived.

The indentation of nested program segments is becoming a common practice. This reduces the problem somewhat, but only to the extent that it serves to visually structure the text. Since this indentation is performed by the programmer, it represents his idea of what the structure is. However, it does not insure that a coding error has not resulted in a different structure entirely.

Furthermore, indentation does not improve locality. For instance, the "THEN" and "ELSE" clauses of an "IF" statement could conceivably be separated by several pages in a program listing. Even if each is prefaced by a descriptive comment (with the resulting problem mentioned above) it is difficult to perceive quickly exactly what is the relationship between the two clauses and what their respective functions are.

It seems to follow from this that an explicitly structured display (using general graphics capabilities) which is guaranteed by the language processor to accurately reflect the structure of the process being designed, should improve readability. If such a display is sufficiently flexible, it can provide that all information relevant to any piece of a program is immediately available, at any desired level of abstraction.

Although a structured display seems possibly useful for improving the readability of programs, still greater benefits might accrue from the increased efficiency possible in constructing and editing programs. This possibility is explored below in the context of discussing the state of the art in program text editing.

Standard linear-text editors are considered first. Then some extensions to text editing functions are explored. Finally, a few experiences with structured text editors are discussed.

Most of the information concerning on-line text editors is due to an excellent survey by Van Dam and Rice (1971). NUROS information is due to personal experience as a user at the University of Nebraska.

Basic Text Editors

Regardless of the fantic form of programming languages, some means must be provided to create a computer - readable copy of a program. Although the development of punched cards was a significant improvement over punching paper tape or manually toggling programs into memory, many programmers prefer on-line text editors.

There is a trend in the computer industry for data entry as well as programming to be done with some assistance from a computer. Several programs have been developed for these purposes. The programs discussed here are those which simply provide a medium for entering and modifying program text.

Typewriter-Based Editors. Most existing program-text editors are designed for use with teletypes or typewriter style terminals. They are strongly oriented to lines of text. In most cases, the standard means of referencing a position in text is by its line number. However, this means is sometimes augmented by string searching capabilities.

QED

One of the earliest of these editors is QED (Quick EDitor), which became available in 1967. QED was created for use on the SDS-930 computer at the University of

California at Berkeley.

Convenience to the user was the primary design criterion (Deutsch, 1967). The command language was designed to be simple and mnemonic, with a text organization that "allows the user to think in terms of the structure of his text rather than in some framework fixed by the system." The intrinsic line orientation was thought to be natural "since all forms of text fall of necessity into lines."

Although the designer submitted to line orientation, he recognized that the use of line numbers for addressing would not be appropriate since it "requires the user to concern himself with an artificial device which has no relevance to his text but nonetheless intrudes on it...." Although text is addressable by line number (which may be altered by insertions and deletions of lines), other schemes are available as well.

If the language to be used with QED includes statement labels beginning in the first character of a line, these labels may be used for addressing. Displacements from labeled records are allowed, so that a reference to :XYZ:-5 affects the fifth line preceding the one labeled XYZ. Any string embedded within a line may be used similarly to reference a record.

Line addressing schemes are implemented by file searches, rather than by any direct means. Each search begins with the current line, resuming at the top of the file if not satisfied by end of file. This means of handling the file thus allows content - directed editing, based on strings appearing in the programming language.

QED also provides some additional conveniences. A line editing mode allows character - by - character editing of a line. Buffers are available for frequently - used commands and for text, and automatic tab stops are provided. In addition to this, the editor saves the sequence of commands on a file, which may be subsequently edited and reapplied to the original file, allowing the reversal of some operations or the maintaining of multiple versions of a file very compactly.

Due to its flexibility and primacy, QED continues to be one of the most popular text editors for use on teletype terminals.

CMS

Another popular editor announced in 1969 is the Conversational Context - Directed Editor. This system, better known as the CMS text editor, was developed at the IBM Cambridge Scientific Center for the 360/67 CP/CMS operating system.

The CMS editor is designed for use with IBM 2741 type terminals, although it has been adapted for use with other terminals.

Program text is stored as 80-byte card-image records. The entire file is maintained in virtual memory as a two - way linked structure. This scheme allows forward - and - back travel in the file, together with line insertions and deletions. This feature has the drawback that any system failure during editing of a file will result in loss of all changes made since the file was last saved.

Much like QED, the CMS editor allows searches and replacements to be specified by indicating a string. In addition, all occurrances of a given string may be replaced by a single command.

Multiple string replacement appears initially to be a nice feature. However, strange things happen when a user intends to replace all occurrances of the variable HI by LOW but forgets that the program also contains several references to the variable NOTHING. Still worse is what happens to a PL/1 program when the string IT is universally replaced by IN, even within the key word BIT.

This difficulty, which is common to most text editors (some exceptions are noted below), is due to the incorrectness of the fantic control. What is generally intended is to rename a variable, but what must be done is to replace a string. Variables do not occur in the fantic space provided by the editor. The result is that the feature must be used cautiously, if at all.

TECO

One approach to solving this problem is encorporated in TECO, a text editor announced in 1969. Text Editor and COrrector is a product of Project MAC at the Massachusetts Institute of Technology. It is designed for use with a teletype and a PDP style computer. It is commercially available for the PDP-10.

TECO allows, and often requires, the construction of a program of basic editing operations. The primitives of this editing language include insertion, deletion, context scanning and conditional execution.

The character pointer used for editing may be positioned within a buffer either absolutely (by numeric value) or relatively (by displacements) or by pattern searches. Special "Q registers" are provided for doing arithmetic and for moving or copying strings. A stored string may be used in turn as a command, so a rudimentary macro facility exists.

Thus, to uniformly substitute one string for another within the file, the file must first be positioned to its beginning (since it processes only in one direction and no backing up is allowed) and then a program must be written. The command to uniformly substitute "GRAPEFRUITS" for "ORANGES" would be J<SORANGES$;-7DIGRAPEFRUITS$>. Perhaps programming aids will become available for programming TECO.

WYLBUR

WYLBUR, another text editor for the IBM 2741 terminal, was created at the Stanford Computation Center of Stanford University. It solves some problems by using absolute line numbers as a stable reference to lines of text, and by allowing operations on _ranges_ of lines.

The use of absolute line numbers solves the problem of providing a stable reference, but becomes inconvenient for the user. A hard copy of the program listing is thus necessitated, in addition to the (otherwise redundant) operation of inserting line numbers. It is just such a requirement which the designer of QED sought to avoid.

The application of editing operations to ranges of lines appears to be a step in the direction of recognizing that the object being manipulated is intrinsically structured. At least the fact is acknowledged that lines might be conceptually grouped. However, it retains the difficulty of using a line - oriented display device which is somewhat slow.

CRT-Based Editors. Experimentation with the use of CRT displays to augment text editors has continued since about 1965 when TVEDIT was developed at Stanford University. This editor differs from those using typewriter type devices in that it allows the simultaneous viewing of several consecutive lines of the current version of the text.

TVEDIT

Editing commands for TVEDIT remain line oriented, but the presence of a marker (cursor) on the screen informs the user which line or character will be affected by the next command. No pattern scanning is provided, and so searches for the text to be modified must be done manually by the user.

NUROS

Another CRT - based text editor is the Nebraska University Remote Operating System (NUROS) which first became available in 1967. The original version was somewhat similar in capability to TVEDIT, but subsequent modifications have improved it somewhat.

Text is edited in either of two modes. The copy mode requires that one file be copied into another while being updated, and is the only mode which allows insertions and deletions of lines. Consequently, it is the only mode for original entry of text. The edit mode allows some reordering and editing of lines in an existing text but no deletions or insertions.

Either mode allows editing of characters within a line by simply overstriking them on the screen of an IBM 2260 (or similar) terminal. String substitutions are permitted only on the screen, but are augmented by a variety of commands in addition to the capabilities of the terminal.

NUROS is basically line oriented, and lines may be addressed by their position in the (output) file. However, a string search facility is also available. Strings may be specified according to content and (optionally) according to their position in the record. A successful search positions the record to the top of the display, enabling editing.

Text Editors with Extended Programming Functions

In addition to those text editors which simply maintain a program text file, there are a number of programs which perform additional programming services. These extra services generally are language dependent and provide help with syntax verification, or documentation.

CCS

Among the earliest of these systems is the Conversational Compiler System (CCS) announced in 1966 (Ryan, Crandall and Medwedeff, 1966). This program is much like the text editors described above, except that it also provides for online execution and debugging.

CCS is strongly line oriented, requiring references to lines in terms of statement numbers embedded within the text. However, editing operations may be applied to ranges of statements, allowing some grouping of related program text. Compilation and execution are online and may be directed to a selected portion of the program. The program is thus also an incremental compiler.

Interactive Programming Support System

A CRT - based system was developed in 1968 by the System Development Corporation. The Interactive Programming

Support System uses the IBM 2260 alphanumeric display to assist with character editing.

In addition to producing and editing a program text, the system performs a syntax check. It is thus language dependent, usable only with JOVIAL.

Editing operations include insertion, deletion, replacement and movement of one or more lines. Editing of characters within lines is performed at the terminal, but must be accompanied by a specification of the line number, since the program does not correlate the display with the text.

Syntax - Directed Documentation

Assistance with the chore of documentation is provided by a system which interactively queries the programmer, based on the structure of the program (Mills, 1970). The system is used with the PL360 language, but is relatively independent of language since it is table - driven.

The query program uses a grammar of the language to identify relevant sections of program text to be described. The grammar, which represents the intrinsic structure of the language, is augmented by special non-terminal symbols which are recognized by the system. Each such symbol is identified with a type of information to be obtained, and thus questions to the programmer are formulated.

Upon completion of the question - answer session, the system produces a report, including the answers given by the programmer together with labels identifying the portions of the program they relate to. The documentation is thus hierarchically organized, according to the block structure of the language. Segments of program text are described at their respective levels of abstraction.

However, these items of information are presented in a sequential list. Due to the mechanism for generating the questions, information tends to proceed from abstract to specific in outline fashion. Nevertheless, no specific effort is made to otherwise augment a reader's perception of the structure, and correlation of the information with the program text is left to the reader.

VERS -- An Incremental Compiler

During the development of a program, it is sometimes desirable to be able to make small changes without necessarily recompiling the entire program. A method for interactively composing and incrementally compiling programs exists (Earley and Caizergues, 1972) for block structured languages as well as unstructured.

In a block structured language (such as ALGOL or PL/1) the addition, deletion or moving of a BEGIN, END or FOR clause can cause major restructuring of the program. Previous systems sidestepped this problem, but the system developed by Earley and Caizergues employs a special auxiliary structure to facilitate limited recompilation.

The implementation requires use of a special language called VERS, whose salient feature is that it restricts the format of program statements to a line - oriented structure convenient for the processor. A "skeleton" data structure is maintained to associate related parts of the program to one another, text to generated code, and symbols to their referents.

This structure (See Figure 1) forms a linear linked list of program statements, linking each to the corresponding compiled code. In addition, every bracketing statement (BEGIN, END, etc.) is linked to the matching statement, thus explicitly representing the block structure of the program. However, this is not a fantic structure since it is only used internally. The actual fantic structure is linear text.

The modification of a statement generally requires recompilation of that statement only. The modification of a declaration requires recompilation of its entire scope. Block brackets (ie. DO, FOR) sometimes require complete recompilation, but the system's claim to fame is that modification of other brackets (eg. IF) affects only the matching brackets.

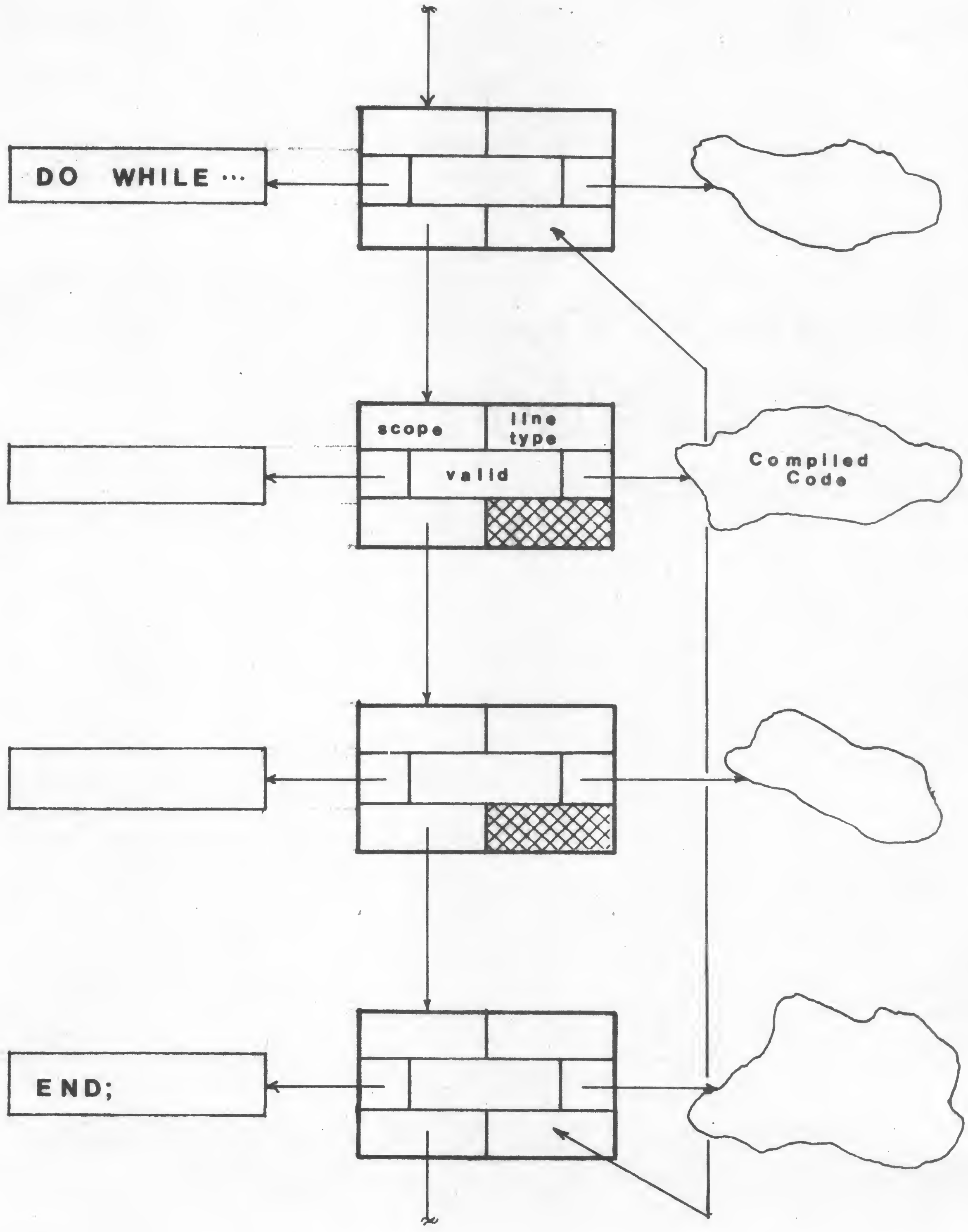


Figure 1: Data Structure for VERS

Editors for Structured Text

The program text editors described above are without exception line oriented. Each faithfully reproduces lines of program text just as entered by the user. The systems described in this section distinguish themselves by their treatment of text as something other than a linear string.

Two of these are explicitly designed for programming. The others are not generally useful for programming, but they manage natural - language text in a manner which is unusual and interesting. Some of the concepts used by WHIMSI derive from them.

Structured Programming Editors. NLS and EMILY are two program text editors which display the program in a structured manner, using CRT type displays. The former is simply an alphanumeric display with automated formatting. The latter uses the graphic and interactive potential of a vector display device.

NLS

NLS is one of the tools employed by researchers at the Stanford Research Institute's center for augmenting human intellect (Engelbart and English, 1968). It is used for creation and modification of programs, reports and research notes.

The principal fantic structure of the system is a tree, with cross indexing of nodes. Each text segment may be named by the user, and the assigned name may subsequently be used elsewhere to reference the segment. The result is potentially a rather complex network of concepts.

Navigation through the text space is thus both vertical and horizontal. Controls are available to regulate the depth of the display. That is, the user may optionally view any number of levels of the tree, beginning at any desired location. When applied to programming, the user imposes a tree structure on his text. The display depth controls then serve to elide lines of the program.

Most controls to the system (including entry and structuring of the text) are implemented with a 5-finger hand set and a mouse. Although all text and control information may be entered with the hand set, an alphanumeric keyboard is available at each terminal. The keyboard gets little use though, since most users find it convenient to keep one hand free to manipulate the mouse, and become quite proficient at using the hand set.

Following years of experience with this system and with the explicit hierarchical structuring of information, Engelbart and English (1968) give a strong testimonial to its usefulness:

> 3c4 The basic validity of the structured - text approach has been well established by our

subsequent experience.

3c4a We have found that in both off-line and on-line computer aids, the conception, stipulation, and execution of significant manipulations are made much easier by the structuring conventions.

3c4b Also, in working on-line at a CRT console, not only is manipulation made much easier and more powerful by the structure, but a user's ability to get about very quickly within his data, and to have special "views" of it generated to suit his need, are significantly aided by the structure.

3c4c We have come to write all of our documentation, notes, reports, and proposals according to these conventions, because of the resulting increase in our ability to study and manipulate them during composition, modification, and usage. Our programming systems also incorporate the conventions. We have found it to be fairly universal that after an initial period of negative reaction in reading explicitly structured material, one comes to prefer it to material printed in the normal form.

EMILY

While NLS allows the user to structure his text, the EMILY system (Hansen, 1971a) requires it. Furthermore, the form of the structure is rigidly tied to the syntax of the programming language used.

In the implementation developed at Argonne National Laboratory, the language used is PL/1. However, the program is table driven and may be used with any grammar.

The fundamental principle of text construction is the simulation of string production from a BNF grammar. Initially, the distinguished symbol (eg. <PROGRAM>) is displayed by the system and is the contents of the program file. Each subsequent addition to the program is the selection from a menu of one of the productions allowed by the grammar for the present non-terminal symbol.

If at any point there is more than one non-terminal in the display, the "current" one is enclosed in a box. All operations in the system apply to the "current" non-terminal or holophrast.

A holophrast is a special symbol which represents a sub-tree of what would be a parse tree for the program. That is, it represents a non-terminal symbol together with all its productions. Its representation is composed of the symbol itself and a truncated prefix of the string that would result from the productions selected.

System controls allow the user to obtain any desired degree of expansion in the display. For instance, the minimum expansion would result in the display of a single holophrast. Greater expansion may display holophrasts, non-terminals, and/or (portions of) a terminal string. Informally, the display at an expansion of n represents the state of the string after n steps of the string production from the "current" symbol.

All editing of the program in EMILY is done in terms of this tree structure derived from the BNF grammar and applied productions. All entry (with the exception of literals and variable names) is by selection from a menu rather than by entry. In fact, only the first use of any variable is entered at the alphanumeric keyboard. Subsequent uses are by selection from a list of variables.

The design of the dialog for programming in EMILY is based on a number of user engineering principles (Hansen, 1971b). Several of these principles are applicable to WHIMSI as well, and are discussed below, in part two.

Hyper-Text Editors. Three text editors which are generally not used for programming, are nevertheless interesting due to their special methods of handling text. Hyper-text (Nelson, 1967) is "the combination of natural language text with the computer's capacities for interactive, branching, or dynamic display ... a nonlinear text ... which cannot be printed conveniently ... on a conventional page...." These systems, HES, FRESS and XANADU, are hyper-text editors.

HES and FRESS

The Hypertext Editing System (HES) and the File Retrieval and Editing System (FRESS) are very similar systems. The latter is an expanded version of the former, commercially available for use on the IBM 360, with nearly any terminal.

The original system (HES) uses an IBM 2250 terminal, and is oriented toward producing "typeset" output on a line printer.

Text is divided into fragments, which are linked into a usable structure by the user. The internal structure of the text is a directed graph whose elements are essentially continuous linear strings. However, this structure is virtually independent of the display or printout, which may be composed from the structure according to the user's instructions.

The light pen and programmed-function keyboard are used to instruct the system in creating and navigating the structure. Navigation is facilitated by the presence of two kinds of links between text fragments.

Unconditional jumps or branches may be placed in the text, causing the display to move directly to the linked fragment. Optional jumps may be inserted, forming a menu of fragments to be visited next.

Fragments may be labeled by the user, and an sorted menu of labels is available for navigation.

Further facilitation of motion through the text is provided by the path memory. Each jump in the structure is remembered, allowing an automatic backtrack.

XANADU

A hyper-text system promised for the future (Nelson, 1973; Nelson, 1974) is called XANADU. It is essentially an elaboration and improvement on HES, a system by the same author.

Among the improvements promised are the ability of the text to move continuously before the viewer, at a rate controlled by the viewer. This provides a degree of screen inertia which helps the viewer avoid disorientation. These features, together with a number of rather nice innovations in fantic controls, are promised sometime in the next few years.

Other Tools

In addition to the on-line text editors described above, several other varieties of programming aid have been experimented with, to varying degrees of success. Those mentioned here are mostly non-interactive. They are mentioned only briefly since their relevance to WHIMSI is marginal.

Decision Tables. The use of decision tables is rapidly becoming commonplace, but considerable experimentation continues toward developing effective processors for them.

A decision table may be informally described as a rectangular matrix of decisions. Along one side is a set of situations, conditions, etc. Along the other is a list of actions to be taken. Each element of the matrix is a Boolean expression indicating whether the action is to be taken for the corresponding condition. In other words, the table is a mapping of conditions into actions.

Typical use of decision tables is to translate them into programs of some high - level language. Current research (Shwayder, 1974) is concerned mainly with methods of producing efficient program logic from limited - entry tables.

Since a decision table provides a clearer picture of a process (in some cases) than the corresponding program text, some research (Cavouras, 1974) has been directed toward converting programs to decision tables. The primary objective is to provide an additional tool for program verification and debugging.

Flow Charts. The use of flow charts is traditional in program design and documentation. Consequently, programming aids have been created which deal with flow charts in a variety of ways.

The simplest of flow chart programs simply constructs a flow chart from the specifications which are provided as input. Although such tools may simplify the process of updating flow charts when programs are changed, they do not insure that it will be done.

Consequently, another variety of program has been designed, which will produce a flow chart from the source program. One example is AUTOFLOW, a product of Applied Data Research, Inc. The vendor's advertisements claim that the program is used in hundreds of computer installations. It simply accepts a program (written in FORTRAN, COBOL, PL/1 or Assembly language) and produces a flow chart.

However, flow charts produced in this manner are little more than boxes drawn around sections of code, with arrows to represent GOTOs. In one case (Weinberg, 1971, p. 265) a programmer was startled to see that the output from his automatic flowcharting program was nothing but his original program neatly surrounded by boxes. The current movement to eliminate the use of GOTO statements will necessarily further limit the usefulness of such programs.

Another approach to the use of flow charts is essentially the reverse process. Some experimental systems allow the user to draw boxes with sections of a program in them, to be constructed by the system into a program. They suffer from the intrinsic difficulty of a flow chart, namely that they do not allow the user to step back and view the program at a higher level of abstraction. Although they do serve to allow direct viewing of the program's flow of control, they do so at such a low level that they are virtually useless except for small programs.

Automatic Program Verification. A number of programs exist which will accept as input a set of program specifications and a source program, and then determine whether the program satisfies the specifications. These programs can generally be guaranteed to verify a correct program *eventually*, but might not terminate if the program is incorrect.

Although verification of loop - free programs is relatively straight-forward, the presence of loops causes difficulties which are the subject of current research.

Some recent developments (Wegbreit, 1974; German, 1975) are promising. Many loops are now handled by the experimental programs. For a survey of general proof techniques, see Elspas, Levitt, Waldinger and Waksman (1972).

Automatic Program Synthesis. Whereas program verifiers attempt to verify a program's correctness according to the specifications, program synthesizers do just the reverse. They attempt to construct a correct program according to the specifications. This process is isomorphic to verification (Lee, Chang and Waldinger, 1974) since a straight-forward algorithm exists to construct a program from its correctness proof.

PART II:

THE WHIMSI CONCEPT

Programs have been developed to help programmers deal with all phases of the design and implementation of a program. Use of a computer to aid in a design process is commonly referred to as Computer - Aided Design (CAD). However, most programming aids now in use do not employ most of the concepts developed in the CAD literature. WHIMSI is an attempt to combine some of the best concepts of CAD systems with those of program design, particularly the use of interactive computer graphics.

CAD is commonly thought to be aimed at relieving the user's drudgery (Hatvany, 1973) by using the computer to perform routine tasks. However, another approach is that the computer can complement the user's capabilities (Meadow, 1970) in a symbiotic relationship. It is the latter approach which is developed here.

Humans have a number of talents which computers do not. The ability to use intuition or creative insight has not been programmed to date. Indeed, it may be argued that the process of improving a computer's insight must necessarily improve that of the designer. Thus it seems reasonable to leave insightful work to humans.

Computers, on the other hand, have a number of talents which humans do not. Humans do not typically perform a million additions in a second. The ability to organize, store, and subsequently to retrieve vast quantities of precise information is more typical of computers than of humans. Furthermore, computers are able to draw pictures many times a second, while the human counterpart may require many seconds to draw the simplest of pictures.

When the insights and creative capabilities of a human are complemented by the speed, accuracy and graphic capability of a computer, a synergistic effect occurs. The result of the interaction is often considerably more effective than the sum of the parts.

One particularly effective system is described by Gero (1973). It is designed for use by an architect in designing a building. Unlike many other systems, it is capable of integrating many helpful programs into one interactive system. Some subsystems merely simulate the effects of a tentative design decision. Others aid in analysing the requirements for the building and their interactions. Gero reports that better designs may be had with less effort due to the use of this system. The effectiveness of Gero's system is due (in part) to its flexibility and its integration of useful tools.

It is not unusual for some CAD system to reduce a month's work to a day. Although not all systems are this effective (some may even limit the user's effectiveness) the existence of systems which are is proof that interactive use of computers can be helpful.

Principal Design Considerations

Although the concurrent requirements of powerful programming tools and flexible methods may sometimes conflict, one simple assumption resolves this conflict in the WHIMSI design. Together with the resulting design criteria. If the design process follows the rules of structured design leads to the design then an appropriate fantic space is feasible. The flexibility of the fantic controls is a direct result of the structure of the fantic space.

Structured Design. It has been demonstrated (Bohm and Jacopini, 1966) that only three formation rules, recursively applied, are required to form any function. These rules are: 1) Sequence: the succession of one process by another, 2) Alternation: the choice of one out of two (or possibly more) alternative processes according to some criterion, and 3) Repetition: Repetition of a process until some criterion is satisfied.

Dijkstra (1972) recommends that only these rules (or possibly some limited superset) be used for programming. This would allow the elimination from high - level programming languages of the GOTO statement (Dijkstra, 1968).

This use of recursive process structures simplifies the construction and comprehension of programs by allowing them to be understood in terms of a small number of well - understood parts. A program is structured by several layers or levels of abstraction. At each level, the process is defined in terms of a small number of processes defined at a subsequent lower level. Construction, verification and comprehension of such a process thus requires little more than that those lower - level processes be constructed, verified and understood. Since each of these latter requirements may be satisfied separately for each sub-process, the structure simplifies the program.

Structured design complements structured programming by motivating its structure (Stevens, Myers and Constantine, 1974). The process of structured design deals with one level of abstraction at a time, limiting interactions by means of functional binding.

All references to any data structure are performed within a single small module (Parnas and Siewiorek, 1975). Since this module may include a set of macro definitions, efficiency need not be sacrificed. Complicated structures may be hierarchically designed in a manner similar to the process, allowing layers of modules to access the layers of the structure.

Although the application of such a discipline to programming and design greatly simplifies the resulting program, it does not limit what can be done. As mentioned above, this is true in principle. However, some programmers question the psychological effectiveness of the discipline, complaining that it is unnatural.

It may be argued that the seeming unnaturalness of this discipline is due to long experience with another; that utility is confounded with habit. To help resolve the controversy, some experimentation (Sime, Green and Guest, 1973) has been performed with non-programmers as subjects. The preliminary results indicate that non-programmers make less mistakes and complete their solution more quickly when they use IF-THEN-ELSE instead of GOTO for implementing decisions.

Fantic Space. Since the process being manipulated is assumed to be explicitly structured in a particular manner, it is reasonable to provide a fantic space which corresponds in form to the structure of the process and the method of designing it.

The inadequacy of linear media for representing a structured concept is discussed above in part one. The choice of a structured graphical representation is motivated by an analysis of the communication process.

Natural - language communication, regardless of the language, makes use of a linear succession of sounds whose implicit structure (Chomsky, 1965) is roughly a tree.

Experimentation at the MIT Artificial Intelligence Laboratory (Quillian, 1968) has produced substantial evidence that this implicit structure corresponds to a subset of the structure of the concept in the speaker's mind.

That this structure is a subset of the actual information to be communicated requires that communication be either incomplete or composed of several utterances representing as many (partially overlapping) subsets of the concept. If the recipient of the information were to have direct access to the original structure, he would be able to explore its contents with less repetition of information.

In other words, a fantic structure is most useful when it conforms most closely to the structure of the concept it portrays.

Thus the display should allow the user to view a (sub-)process at any level of abstraction, with descriptions of its sub-processes and a graphic portrayal of their functional relationships. For instance, a process subdivided according to alternation should appear different from one which is simple succession.

WHIMSI divides the program into <u>segments</u>, each of which describes a process at some level. Except at the lowest level, each segment is superordinate to one or more other segments. Each is associated with an English description which describes the function of the process at the current level.

Since the resulting tree structure may not be entirely rich enough to support all operations the user might desire, special provision is made for arbitrary named links between segments. Floating links also exist, which allow direct access to a specified segment from any part of the structure.

In principle, analogous structuring facilities should be provided for the program's data base. The means to provide this facility remain the subject of continuing research.

The fantic space is supported by a structured data base which represents the process under design. Each segment is an element of the data base, and all segments are stored on a disk dataset, referenced by name. In principle, the actual machine instructions should be associated with their corresponding segments, providing the actual structure for execution.

The problem of reference during editing and execution has been solved in the current implementation, but only at some loss in efficiency. This loss may be prevented by the use of capability - based addressing (Fabry, 1974).

This is an addressing scheme originally proposed to help solve some problems in computer security. It requires that every reference to an object be mediated by a "capability" which describes its attributes and location.

Since the location is included in this description, the system can automatically retrieve it from permanent storage for use. This is somewhat analogous to virtual memory, but avoids the limitations of a fixed, arbitrary page size.

Fantic Controls. The principal objective in the design of fantic controls is to maintain adequate flexibility and power to obviate the use of hard-copy for the entire programming task.

All controls are based on the segments and links of the fantic structure. The entire process of programming is performed by use of a set of operations on these elements and the strings attached to them.

As such, WHIMSI is not only a medium for program entry, but is the framework for a number of other programming aids. As explained below, it is possible to incorporate a complete set of programming aids into the WHIMSI structure. Doing so is economical in that it saves the duplication of parsers and flow analysis routines that are used by many programming aids.

Integration of Programming Aids

Programming aids include not only entry and update, but a variety of other functions as well. Synthesis and verification are complementary functions with which some computer help is desirable. Documentation is also essential to the normal use of programs, and so its integration into a programming system is desirable as well. Likewise, optimization and incremental compilation of programs are part of a complete system.

Each of these may be easily incorporated into the WHIMSI framework. Indeed, such incorporation is relatively easy and natural. Some functions may even be improved by the integration -- a synergistic effect among programs.

Documentation. A complete and accurate description of the workings of a program is generally essential to its long - term success. Users must know how to make it do what is desired. Other programmers may need to know how to modify it.

The needed descriptions are naturally incorporated into the program as it is being constructed under WHIMSI. Since each segment requires that a description be attached, that description may be used as effective documentation.

This mode of documentation has a distinct advantage over the usual documents. Since descriptions are attached to each segment, the program is described at every level of abstraction, and the relevant document is always displayed with the appropriate portion of the program.

Furthermore, the intrinsic nature of WHIMSI is to display the structure of the program. This obviates the use of flow graphs and other documents which explain the program logic.

Synthesis and Verification. When linear media are used, a program is typically written before it is entered into the system. However, with a structured medium such as WHIMSI this is not necessarily true. Indeed, it is possible to incorporate routines which will aid in the process of on-line design.

The use of recursive structures in a process facilitates the analysis and design of that process. The programming process itself may be axiomatized and at least partially automated (Mills, 1975). Experimentation at the MIT Artificial Intelligence Laboratory (Sussman, 1973) has demonstrated a great reduction in the complexity of problem solving (especially programming) which is due largely to a top - down structured approach.

Dijkstra (1972) indicates that "useful structure" can simplify the job of construction (as well as verification) of programs. Consequently, if the structure of the program is immediately available to a verification program then its job will be simplified.

Furthermore, the interactive and display environment maintained by WHIMSI can provide a sound foundation for the construction of interactive verification routines. Again, the synergistic effects of man - machine cooperation can be exploited.

Even when verification routines are not employed, debugging can be facilitated if a program failure produces a pointer to the failing segment. This would allow the programmer to examine the failing routine at any desired level of abstraction. This differs from most programming systems which allow analysis of the error only from limited diagnostics or (in the case of a dump) at the machine level.

Incremental Compilation. The maintenance of large, integrated program systems can be augmented by the ability to recompile only those portions of a program affected directly by some change. As was demonstrated by the VERS system described above, explicit recording of the program's structure greatly improves the efficiency of incremental compilers.

Program Optimization. Once a program has been completed and is working as desired, it is often desirable to enhance its efficiency by means of some optimizing transformations.

The art of program optimization is well developed, but the one thing all optimizers have in common (Schaefer, 1973) is that they require an analysis of the program flow. Since this analysis is explicit in the WHIMSI structure, the job of the optimizer can be simplified.

Furthermore, it generally occurs that the majority of a program's execution time is limited to a small portion of the program (Knuth, 1974). In such a case it is pointless to attempt improvement of the greater part of the program, since such improvement would most likely not be noticeable. If it can be determined which portions of the program must be improved, selective optimization is desirable.

This can be facilitated within the WHIMSI structure by attaching execution monitors to various segments and by providing a fantic control to direct optimization at segments selected by the user. Indeed, the optimization may be performed interactively, under control of the user to any desired extent.

The Current Implementation

The original conception of WHIMSI was as a total programming environment, complete with its own means of generating machine code. However, the available time for implementing it required several compromises in the design.

The limitations of this prototype are further increased by the unavailability of sophisticated hardware and software. The best of available graphics devices at this installation is an obsolete device: the IBM 2250 Model 1. The only high - level software available for the 2250 is based on FORTRAN subroutine calls. It is bulky and inefficient, and not very suitable for dealing with recursively - structured displays.

Consequently, much of the available time was spent developing basic software and simulating the needed hardware.

General Description. Due to the lack of time, the idea of generating machine code directly through WHIMSI was abandoned. Instead, the current implementation is a pre-processor for the PL/1 compiler. That is, it presents something resembling the desired fantic space to the user, and then translates the user's program into PL/1 code.

Since the resulting system is not a language processor, no parsing is performed. Consequently, there is no syntax checking and no possibility of including any special programming aids. All that is provided is a structured medium for creating and modifying programs.

A single tree - structured program may be constructed in a file. In addition to the tree structure, there are links as described above. The resulting multiple structure is the basis for the acronym: WHolly Interactive Multiply Structured Information (WHIMSI).

This system (which is more fully described in part three) is severely limited. Indeed, it is not expected to be generally useful as a programming tool. However, it is effective for evaluating some of the basic concepts involved. Specifically, it provides a basis for testing the effectiveness of the programming methodology, and of the graphics used. It also provides a means of observing the sequences of actions used by a programmer.

Although evaluation cannot be complete with an incomplete system, some valuable information can be obtained. Since this information will necessarily affect the design of a complete system, it is nevertheless useful. A summary of what has been learned to date from use of the system is included in the Conclusions.

User Engineering Considerations

The principal considerations in user engineering, according to Nelson (1973), are "making things look good, feel right, and come across clearly."

Recognizing that the user is human is of prime importance, according to Hansen (1971b). Thus the designer should keep in mind that "the features are being designed for the user rather than the other way around. In fact though, any interactive system will require retraining of the users and some systems ... may require the user to alter thinking habits of many years standing."

Some salient aspects of humans which must be considered are that they have short memories and are apt to make mistakes. Moreover, most humans like to work at some particular rhythm and become impatient or irritated if the system's response time disrupts that rhythm.

Thus, the three principal design objectives of WHIMSI are simplicity of use, fail - soft behavior in case of mistakes, and a good response time.

Simplicity. Since the use of an interactive system involves both intellectual and manual activity, the system should be adjusted to be simple for both.

Conceptual simplicity in WHIMSI is obtained by limiting the concepts for operation to a small set. The rules are few and simple. They apply uniformly to all cases which the user may encounter. Any necessary exceptions are fully specified and hopefully easy to remember. In any case, the system reminds the user whenever an exception occurs.

Further simplicity is obtained by not requiring the user to memorize a list of key words or symbolic command words. Instead, all operations are performed by means of the programmed function keys or the light pen. However, the alphanumeric keyboard is used for entry of program text. This is necessitated by the fact that the system does not interpret the program and so must treat it as simply a string of characters.

Manual simplicity is obtained by designing the command structure to minimize the number of steps required to perform any action. Where simplicity must be compromised, it is only for infrequently used operations. The principle of selection (as opposed to entry) of commands speeds operations by requiring only a single key press to perform an operation.

<u>Fail - Soft Features.</u> Since human users (and sometimes the operating system or computer operator) can make mistakes, an important consideration is the minimization of the impact from mistakes.

The first step toward this objective is to minimize the likelihood of errors occurring. Simplicity of operation helps. Further help derives from the prompting provided by the system. The programmed function keyboard indicates at all times those keys which may be used next, by illuminating them. Each discrete action by the user (ie. key press or light pen use) is prompted by a separate query from WHIMSI.

When an error does occur, and the system detects it, diagnostics are produced which briefly explain the reason that the action was rejected. The only exception is when an improper key is pressed or the light pen is used in a manner not responsive to the query.

In these cases, it is assumed that the key was pressed accidentally or that the pen was used inaccurately. Thus the user is not intimidated by an undeserved diagnostic; instead, the action is ignored. The lack of action on the part of the system is sufficient indication to the user that the response is not accepted. In the case of light pen, this feature simply simulates a failure of the 2250 to detect the pen. In the case of the keyboard, it simulates turning off the key.

When no error is detected by the system, the action indicated is taken but the user has the option to reverse it after seeing the result. In case of some operations which may cause major damage to the file if done improperly, the user is reminded of what will occur and asked whether the operation should be completed. This allows multiple options for backing out of a bad action.

WHIMSI also protects the user from the operating system. If a system failure occurs while the file is being updated, or if the program is abnormally terminated for any reason, only the most recent changes to the file will be lost.

The portion of the file being modified is always kept in core. From time to time, the updates are stored on the disk file rendering them permanent. At any time before this occurs, the user may reverse the changes by restoring the core image from disk. Furthermore, the procedure for moving updates to disk is carefully designed so that, if it is interrupted for any reason, the structural integrity of the file is not compromised. Instead, the updates are lost.

Response Time. Although response time may vary considerably with the operating system's load, it is generally fairly good. The importance of response time is described in depth by Martin (1973) and by Hatvany (1973).

Martin reports research which indicates that the maximum acceptable response time varies with the type of operation. The inference is that a longer response is acceptable when closure is obtained than when it is not.

Hatvany reports an instance when the same interactive system operating on two computers received strongly opposing ratings from the two groups of users. He traced the difference to the fact that on one computer the response was nearly instantaneous, while on the other there were delays averaging just less than one second.

In order to obtain as quick a response as is feasible, the system is designed to avoid operations which may require long or varying periods of time to perform. Thus there are no commands which require a search of the data base. Instead, the data base is designed so that searches are not needed.

<u>Other Factors.</u> Although factors other than those mentioned here are important to adjusting the system to the user, these are the most important in the current design.

Some factors such as keyboard arrangement and design are not programmable and so the existing hardware must be tolerated as found. For instance, the physical separation of the function keys from the alphanumeric keyboard is not convenient.

Furthermore, the pressure required to activate a function key is excessive, and the delay of response until the release of the key is not desirable. The absence of a new-line key on the alphanumeric keyboard is a major defect since it must be simulated by a programmed function key under program control.

One advantage of the programmed function keys is that they accept an overlay which labels the keys. Considerable attention was paid to the design of an effective arrangement of keys.

The keys are grouped according to related functions, to the extent that is possible. Varied orientations of groups of keys, together with the use of colors and the illumination of keys, helps the user locate the desired key for any function. Some physical separation of groups of keys is made possible by the fact that only 19 of the 32 keys are used.

PART III:

USING WHIMSI

WHIMSI is designed for easy use by the experienced user, and for quick learning by the beginner who has some programming experience. The few simple concepts included here are thus sufficient to enable the use of all the system's features.

Description of the system is divided into three parts. The first part presents the structure of the fantic space and the second enumerates the fantic controls available. A final section discusses some special ideas and hints for programming methodology to be used with WHIMSI.

The presentation of the fantic space includes the division of programs into segments and the descriptions of the types of segments available. Links, floating links and chains of segments are also discussed in terms of their ability to provide flexible cross referencing and look-up power. This section also discusses the display conventions used.

Discussion of fantic controls includes a description of the two keyboards used, and a general description of the use of the 2250 terminal. The principal semantic elements of the command language are explained, followed by the overall structure of WHIMSI operation. Specific controls are finally explained, including a detailed description of the interactive protocol.

The section on programming methodology simply puts forth some habits which have been found to be useful for programming with WHIMSI. Although they may be different from normal programming habits in some respects, they are worth consideration by any user.

The Fantic Space

All displays produced by WHIMSI, and all actions by the user, are mediated by the structures of the fantic space. The principal elements of the fantic structure are the program segments, which form the basic tree structure. Arbitrary links between segments, and floating links to segments, together with the chaining of segments, provide cross referencing within the basic structure.

All elements of the fantic structure are displayed according to their special conventions. Each display element effectively provides a handle for the user to manipulate the corresponding structure element.

Segments. A program in the WHIMSI system is a segment with zero or more subordinate segments. Each segment represents an operation at its respective level of abstraction.

The information associated with each segment is its type, its abstract natural-language description and a list of its subordinates. The description is the first part entered by the user, and provides a guide for completion of the program as well as some documentation.

Six types of segment appear in the system, of which five appear in a completed program. The "undefined" segment appears in a program under construction to designate a location where a segment is to be placed. Since the undefined segment is incomplete, it includes information only concerning its type and the abstract description of the segment which is to replace it.

The remaining five types are Catenate (CAT), Repeat, IF, Text and Null.

A Catenate segment has two subordinates. As its name implies, one segment is catenated to the other, following it sequentially in time. CAT may thus be used in conjunction with other segment types to form them into a sequential process.

The Repeat segment also has two subordinates. However, unlike CAT, the first of these is not a segment but a string. The string specifies the parameters of the repetition, and can be any string which may follow "DO" in PL/1. Note that no semi-colon (;) is required, and that the DO is not written by the user since it will be supplied by WHIMSI during creation of PL/1 code. The second subordinate of the Repeat is a segment which represents the scope of the repetition.

The IF segment has three subordinates. Much like Repeat, the first of these is a string. In this case the string may be any Boolean expression allowed in PL/1. Note that the semi-colon (;) <u>must not be coded</u>. This string will become part of an IF statement in the PL/1 code generated.

The other two subordinate segments represent the scopes of the THEN and ELSE clauses respectively. If either of the clauses is not required by the program, a Null segment should be entered in its place.

The Null segment may be coded anywhere, but is especially useful for balancing a one-sided IF. Its function is to complete the structure without generating any program code. Unlike other segment types, the only information associated with it is its type. No description is needed.

The bottom-most level of the WHIMSI hierarchy is the Text segment. As its name implies, its associated information includes some PL/1 text. Although the abstract description is also present, there are no subordinate segments. Text which appears in this segment will appear in identical form within the PL/1 program produced.

Cross References. Although all information relevant to the actual functioning of the program is contained in the segments, additional information which may be useful to the programmer is included by means of cross references. These take the form of links, floating links, and chains.

A link is a special path provided between two segments. It has a definite beginning at one segment and is directed toward another. A name is provided by the user for each link, allowing easy recognition. Links provide a means to reference a segment which is not currently displayed but which is related in some (arbitrary) manner to the current segment.

Floating links are similar in that they are directed toward a particular segment. However, they are not rooted at any particular segment and thus "float" around, following the display. Consequently, a floating link allows the user to reference a particular segment regardless of the current location of the display.

A chain provides a continuous path through a sequence of segments determined by the user. It is composed of a set of links arranged in such a manner that each segment of the path includes a link to the next. An optional floating link to the first segment provides easy access to the chain at any time. The special linkage (See Figure 2) of a chain provides for easy insertion and deletion of segments.

Although chains may be maintained by the user, a special chain is also maintained automatically by WHIMSI. Its name is "Undefined Segment" and, as the name applies, relates to those segments which have not yet been defined. Each time an Undefined segment is defined (by replacing it with another segment type) another Undefined segment may be created subordinate to it. When this occurs, the new segment is inserted in the chain. In any case, the newly defined segment is deleted from the chain. Thus, during program construction, the user has immediate access to those

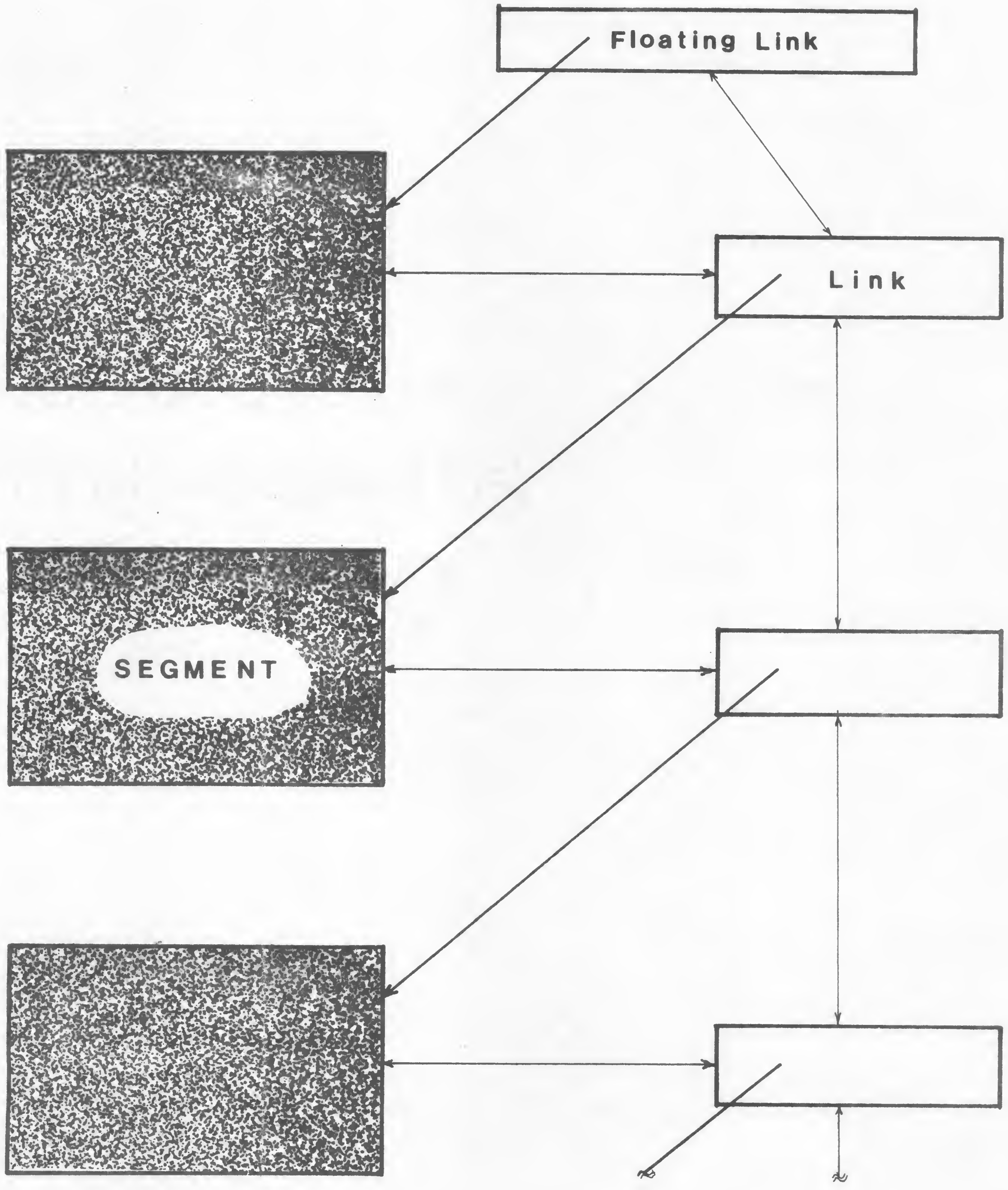


Figure 2: A Chain of Segments

segments which remain undefined. This provides the option of deferring definition of a segment without fear of forgetting where it is. Note that when the program is completed the chain disappears.

Display Conventions. Each element of the fantic structure is displayed according to the appropriate conventions. Thus the display includes segments and links. Since the entire structure cannot be displayed at once, some of the conventions deal with what is to be displayed, as well as the formats.

The principal division of the display is into segments and links. The top line of the screen is reserved for queries from WHIMSI. Links, floating links and chains are combined into one group for display since their properties are somewhat similar.

At any given time (while viewing the program and not editing a string) all floating links (including those which are parts of chains) are displayed. Only those links associated with the current segment are shown.

Each link or floating link is displayed as a single line of characters which is its name. The first link for the current segment (if it exists) is displayed as the bottom line on the screen. Subsequent links, followed by floating links, occupy subsequent lines above the first. The remaining portion of the screen is used to display the current segment and zero or more of its subordinates.

Each segment (with the exceptions of Null and Undefined) is displayed as a box with its type indicated at the top (See Figure 3). The segment's type is further distinguished by additional lines within it. The first line of the segment's description is included immediately above the box.

The Null segment is displayed simply as a large "X" in the middle of its allocated space. An Undefined segment appears as "?" below the first line of its description.

The Text segment is simply a box (marked TEXT) with the text displayed within. Any lines of text which exceed the width of the box are simply continued on the next line. Lines which cannot be thus included within the height of the box are omitted. However, a full frame view of any string (including text) is available, as described under the heading of fantic controls.

The CAT segment is subdivided horizontally by a single line. The flow of control is implied from the upper segment to the lower. The Repeat segment encloses the repetition control clause within a small partition at its top; the remainder of the box contains the repeated segment. The IF segment contains the Boolean expression in the triangular partition at the top and the THEN and ELSE clauses are separated by a vertical partition, in the lower part of the box.

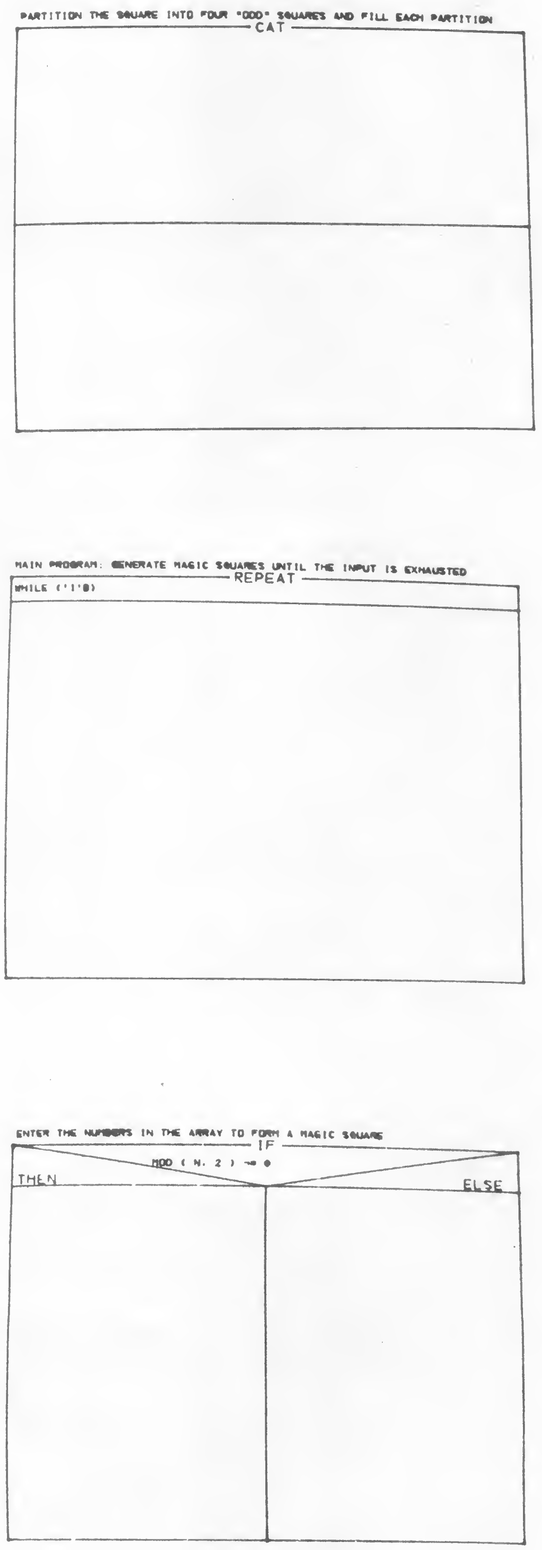
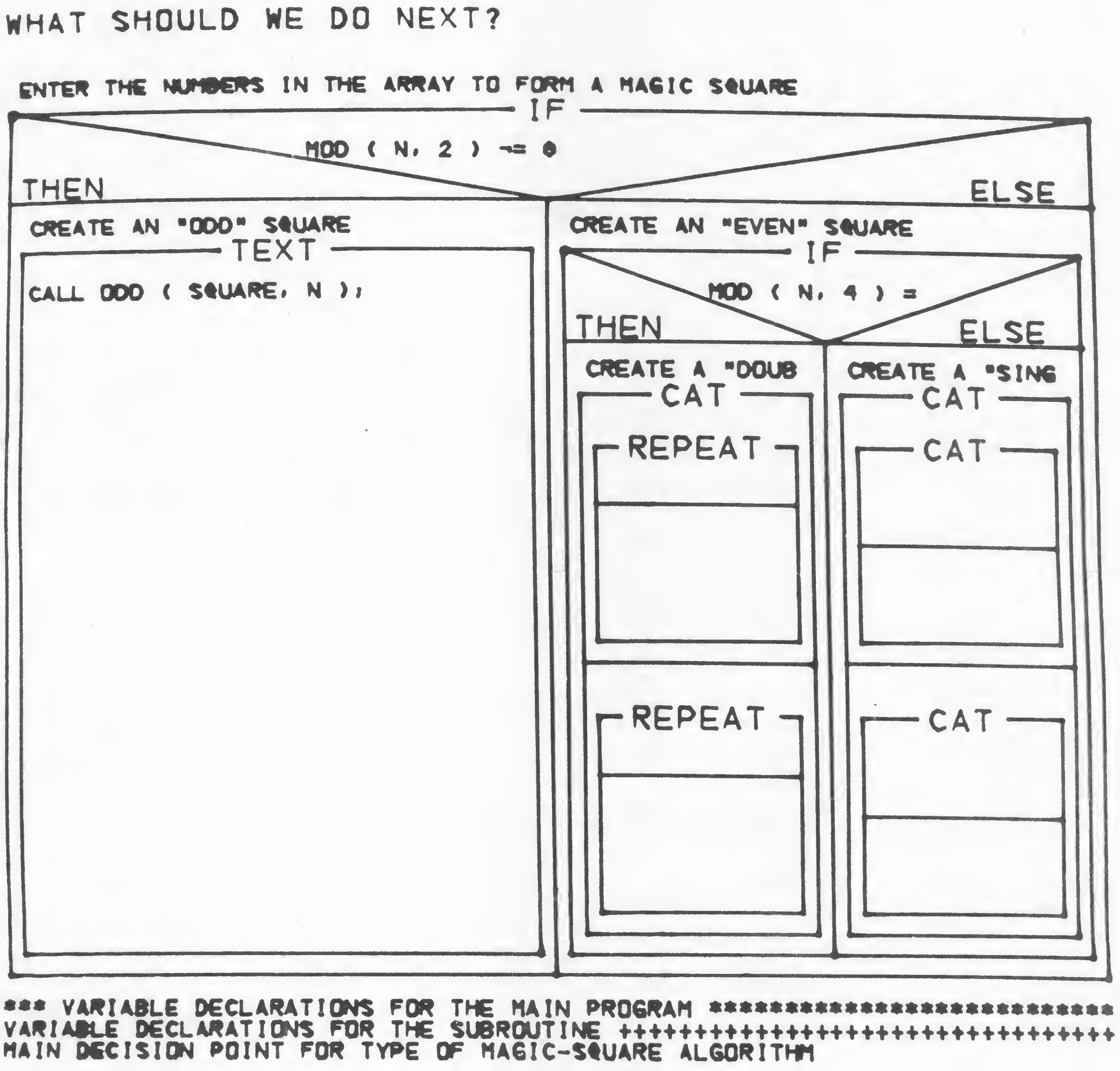

PARTITION THE SQUARE INTO FOUR "ODD" SQUARES AND FILL EACH PARTITION
CAT
MAIN PROGRAM: GENERATE MAGIC SQUARES UNTIL THE INPUT IS EXHAUSTED
REPEAT
WHILE ('1'B)
ENTER THE NUMBERS IN THE ARRAY TO FORM A MAGIC SQUARE
IF
MOD (N, 2) ¬= 0
THEN
ELSE

Figure 3: Segment Display Formats



```
WHAT SHOULD WE DO NEXT?

ENTER THE NUMBERS IN THE ARRAY TO FORM A MAGIC SQUARE
MOD ( N, 2 ) ¬= 0
CREATE AN "ODD" SQUARE
CALL ODD ( SQUARE, N );
CREATE AN "EVEN" SQUARE
MOD ( N, 4 ) =
CREATE A "DOUB
CREATE A "SING

*** VARIABLE DECLARATIONS FOR THE MAIN PROGRAM ****************************
VARIABLE DECLARATIONS FOR THE SUBROUTINE ++++++++++++++++++++++++++++++++++
MAIN DECISION POINT FOR TYPE OF MAGIC-SQUARE ALGORITHM
```

Figure 4: The WHIMSI Display

Any segment which has subordinate segments (ie. CAT, Repeat and IF) thus includes one or more areas within its box within which the subordinate segments may be displayed. The display is thus recursive, to a depth controlled by the user (See under fantic controls).

The resulting display (See Figure 4) gives an immediate view of the structure of that portion of the program being viewed. Furthermore, it provides (in conjunction with the fantic controls) a simple means for moving the display to any desired portion of the program structure. Thus the user need not be concerned with hard-copy program listings.

The Fantic Controls

The fantic controls provided by WHIMSI are implemented by means of a programmed function keyboard and a light pen. Additional input is allowed through the alphanumeric keyboard. This section provides the basic instructions for using these controls to construct a program.

Discussion begins with a brief description of the IBM 2250 display unit and its general operating instructions. This is followed by an exposition of the principal semantic elements of the command language. A description of the overall structure of the program's use follows, followed in turn by the specific commands which may be used.

The IBM 2250 Display Unit. The terminal used for the initial implementation of WHIMSI is the IBM 2250 Display Unit Model 1. A complete description of the device is included in the IBM component description (Form GA27-2701).

The principal elements of the display unit are a CRT vector display, a buffer for regeneration of the display, a light pen, a programmed function keyboard and an alphanumeric keyboard.

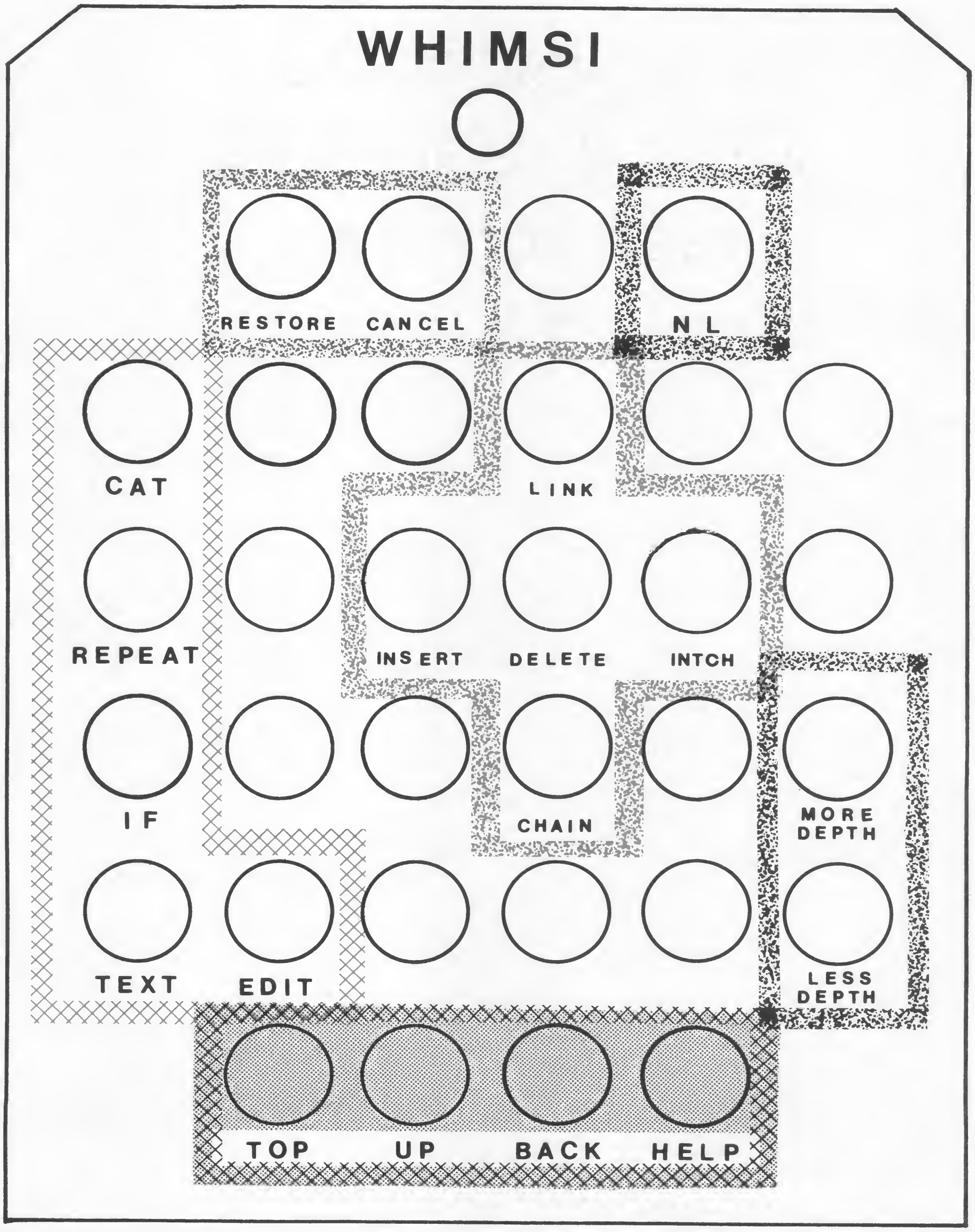
The CRT has a face which is twelve inches to a side. These twelve inches are divided into 1024 raster units, which are the basic units of screen addressing.

Hardware vector and character generation are available, under control of the regeneration buffer program. Two sizes of characters are available, the larger of which is half again larger than the smaller. The display can accomodate up to 52 lines of 74 small characters each. Characters to be displayed, together with vector and positioning information are transferred from the main memory to the 2250 buffer over a selector channel under program control.

The light pen is enabled by a foot switch and may be used to indicate anything which appears on the screen. When the pen is activated and detected, regeneration of the image is halted and the buffer and screen locations of the pen detect are made available to the program, which may optionally restart the regeneration.

The programmed function keyboard consists of 32 keys positioned somewhat to the left of the alphanumeric keyboard. Each of the keys may be separately illuminated under program control. The pressing of a key interrupts the processor, making the action known to the operating system. The key number is subsequently made available to the user program, which may take appropriate action. An overlay (See Figure 5) may be placed over the keys to provide labels.

The alphanumeric keyboard (Figure 6) is available for text entry. In WHIMSI it is used only to enter program text (in Text segments) and the other strings (including descriptions and names) which are required. Although it is


WHIMSI
RESTORE
CANCEL
NL
CAT
LINK
REPEAT
INSERT
DELETE
INTCH
IF
CHAIN
MORE DEPTH
TEXT
EDIT
LESS DEPTH
TOP
UP
BACK
HELP

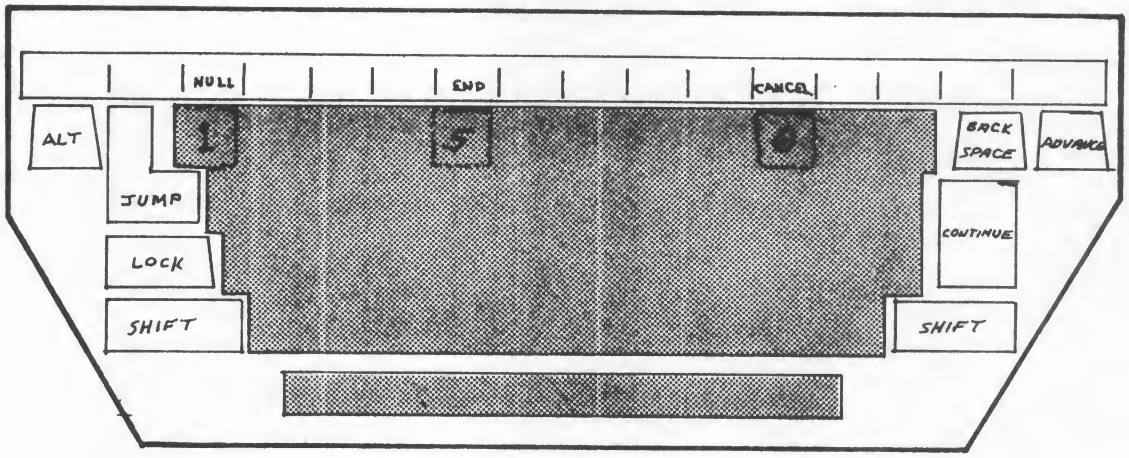


Figure 6: Alphanumeric Keyboard

similar in most respects to most typewriter keyboards, it is different in some important respects.

The keyboard becomes active only when a cursor is displayed on the screen (Exceptions are noted below). If a key is pressed at any other time it will lock and must be manually freed. This is done by holding down the Shift key and then pressing Alt.

The space bar types a space and will consequently destroy whatever information is over the cursor. Non-destructive advancing and backspacing of the cursor are provided by the Advance and Back Space keys respectively.

When it is desirable to delete a character from a string without replacing it by a space, the Null character may be typed. This is done by holding down the Alt key and pressing Null (Alt "1"). This enters a null character in the buffer, causing the overstruck character to disappear and the surrounding characters to close on it. The terminal does not display the null character and acts as if it weren't there. Consequently the cursor will not be visible when positioned on a null character. It is nevertheless effective and characters may be entered over the nulls.

Any of the keys mentioned above, in addition to the alphabetic, numeric and special-character keys, may be used as repeating keys. If the Continue key is depressed, any other key will repeat as long as Continue remains depressed, even if the other key is released. This allows rapid removal of strings by repeating Null, the padding of strings by repeating (for instance) the asterisk, or rapid cursor motion by repeating Advance and Back Space.

If at any time it becomes desirable to move the cursor to the first character of the string, this may be done by pressing the Jump key. Although the normal function of this key is to move the cursor to the next string, WHIMSI never displays more than one string when a cursor is present. Thus the action wraps around to the beginning of the string.

Two other keys are of special interest. They are the End (Alt "5") and Cancel (Alt "0") keys. The End key does not affect the string, and thus may be used whether a cursor is present or not. It provides an attention to the program and is interpreted by WHIMSI exactly as the Back key on the programmed function keyboard. This duplication of functions is under program control and is provided for the user's convenience during some operations involving the alphanumeric keyboard.

The Cancel key is not used by WHIMSI since it is confiscated by the operating system. However it must not be pressed under any circumstances since it will cause WHIMSI to abnormally terminate. Its normal functioning allows resumption of the program by the user, but the manner in which the terminal is controlled by WHIMSI does not allow the program to resume. Its function is replaced (as described below) by a programmed function key. Note that this key should not be confused with the key marked Cancel on the programmed function keyboard, which serves an entirely different function.

Semantic_Elements. Apart from the actual typing of character strings, which is done at the alphanumeric keyboard, each element of the command language is a single key press or a light pen detect. These elements consist of verbs and nouns.

All keys of the programmed function keyboard are verbs in the command language. Each indicates a specific type of action to be performed. In some cases these verbs are augmented by one or more nouns which specify the object(s) of the action. Since not all verbs are valid at all times, those which are acceptable at any given time are illuminated for the user's benefit.

The nouns in the command language are provided by the light pen. In general, the object is pointed to. Although any use of the light pen is a noun indicating the object pointed at, some nouns when used alone (ie. not preceded by a verb) are treated as verbs.

Objects which may be pointed to by the light pen are grouped according to type. In some cases, a menu is displayed on the screen together with a query. Each item of the menu is thus a noun. Since no other form of command is permitted at a menu frame, the total information conveyed to the system is the choice from the items.

A segment is indicated by pointing to its containing box. That is, the box or (preferably) its type indicator is interpreted as meaning the segment. In the case of a Null or Undefined segment, the single character displayed ("X" or "?") identifies the segment. In some cases only a "structurable" (ie. Undefined) segment is acceptable. It is indicated in the same manner.

Another means of indicating a segment, which is not available when a structurable segment is required, is by pointing to a link which references it. Since segments and links may never be referenced in the same context, no ambiguity occurs. The prohibition of link references to structurable segments is due to the use of such segments, which is replacement by a structure. Since this may strongly impact the overall file structure, the user is required to view the structure before modifying it so that errors are less likely.

As mentioned above, pointing to a link (when a link is appropriate to the context) indicates that link.

A string is indicated by simply pointing to it. In the normal display mode, strings include segment descriptions, text and link names. Although the query line and segment type indicators are strings they are not recognized as such semantically.

Note that an ambiguity may occur due to the multiple use of the link. Although the segment - link ambiguity is resolved by the mutual exclusiveness of the appropriate contexts, the segment - string ambiguity is not. Consequently, it is resolved in favor of the segment interpretation. This does not cause any serious problems.

The remaining item on the list of noun types is the character. Any character within a string may be indicated by simply pointing to it. This does not cause ambiguity since characters and strings are never mentioned in the same context.

<u>The Basic Command Structure.</u> All commands given to WHIMSI by the user follow the same syntactic form. Each is a single verb (sometimes a lone noun) followed by zero or more nouns. Each element of a command is prompted by a query from the system, so that sequences need not be memorized.

In addition to those keys used for the normal entry of commands, there are two additional keys which may alter the sequence.

The first of these is the Cancel key of the programmed function keyboard, not to be confused with the key of the same name on the alphanumeric keyboard. This key generally allows the user to cancel a command (or part of it) before it is completed. Since no action is taken by the system prior to completion of a command, this feature allows the user to change his mind. In some cases an error diagnostic requests that the user Cancel the command to acknowledge its rejection by the system.

The HELP key is of primary importance since it allows the user to interrupt any operation with a request for information or program termination. Since it is always activated (except, of course, when in HELP mode), the HELP key may be used any time the system is requesting any action. The user may subsequently resume the interrupted operation exactly where he left off, since complete status is saved.

The Commands. As mentioned above, the HELP key may interrupt any operation. However, it is actually a part of a comprehensive means to navigate the fantic space.

NAVIGATION

Navigation through the WHIMSI space may be considered as consisting of two independent subsystems. A global system allows free movement between two files and the HELP frame. The local system provides for navigation within a structured file.

The global environment of WHIMSI consists of two files and the HELP frame. It is the latter which appears first

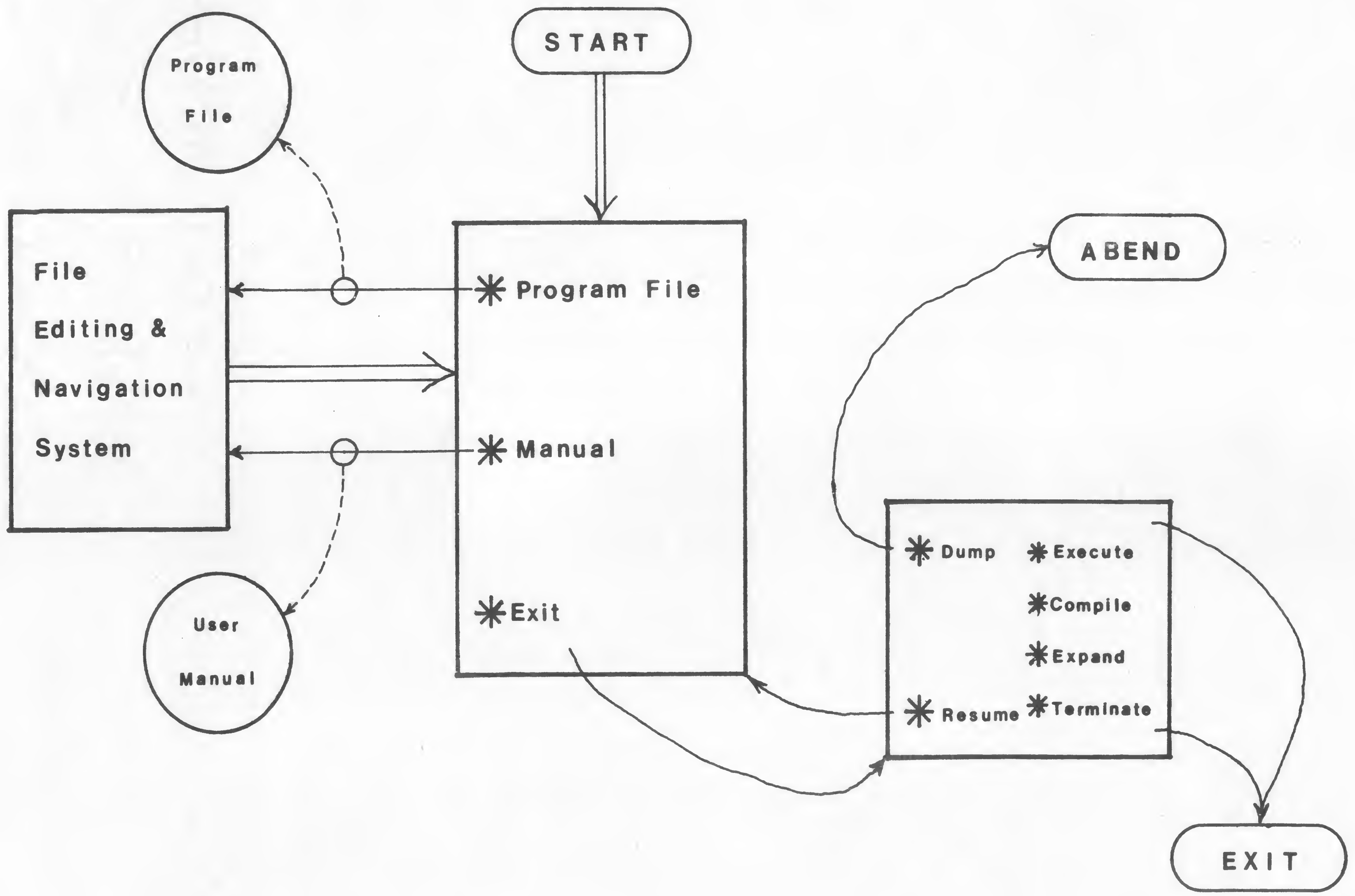


Figure 7: Global Navigation Paths

upon beginning use of the system. It is composed entirely of a single menu with three options.

The first two options are "Program File" and "Manual" respectively. Each of the two is a structured file. The first is the princpal file on which the program is to be constructed. The other is a file of similar structure which includes a special on-line version of the operating instructions, complete with cross references and indexing.

The third option is "Exit" which brings up another menu. This menu includes a "Resume" option (which returns the display to the HELP frame) and five termination options. The first option is "Dump" which causes the program to abend with a User 0000 completion code. The other termination options set the condition code and terminate WHIMSI. Under the control of appropriate JCL (See Appendix I) the return code may permit conversion to PL/1, compilation, and execution of the program. Alternatively the job may be halted before any of these. Thus the user has options covering the range from simple discontinuation of editing to a run of the program.

The HELP frame itself may be entered (as mentioned above) either from the Exit frame or from any point during editing of either file. Selection of either file from the HELP menu returns to the point in that file from which HELP was entered.

FILE NAVIGATION

Embedded within a WHIMSI file are a number of navigation aids, each of which is constantly displayed for the user's convenience.

The presence of segments (and links to segments) on the screen allows the user to move the display to any of several segments by simply indicating it with the light pen. In this case, the entire command is composed of a single noun which designates where to go.

The display may similarly be moved to a full frame view of any string by simply pointing to it. The only exception is the string which is the name of a link, which is always entirely visible anyway. When display moves to a string, the user is given the option immediately to edit it or to return to the normal display by pressing Back.

Although display by selection works well for moving down the tree (or to predetermined locations via links) the upward motion in the structure is conveniently done by the Up key. This key simply moves the display to that segment which is immediately above the present segment. In the case that the display is already at the top of the tree, no action occurs.

If the top (or root) of the tree is desired, it may be reached by simply pressing Top. This command resets the view to the root of the tree and erases the memory (See below) of the preceding path.

When any of the above means is used to move the display within the structure, a record is maintained of the location just left. This information is kept in a push - down stack and thus may be used to back up along the path. Use of the Back key (or its equivalent, "End") accomplishes this, popping the stack.

This feature is particularly useful when a link has been followed to some segment and a return is desired. It is also recommended (when appropriate) as an alternative to Up since use of Up tends to increase the depth of the stack needlessly. The Up command is thus most useful when a link has been followed to a segment and it is desired to observe its context. It is also useful when the user wishes to move up to view a segment in its context and then return to the original display.

These few commands comprise the entire navigational system. However, another aid is provided which facilitates their use somewhat.

Since the recursive structure of the display may sometimes become cluttered, the depth of display may be controlled by the user. Depth of display varies between zero (only the current segment is visible) to three. The More Depth and Less Depth keys increase and decrease the display depth, respectively. In the case that an increase (or decrease) of depth might cause the bounds to be

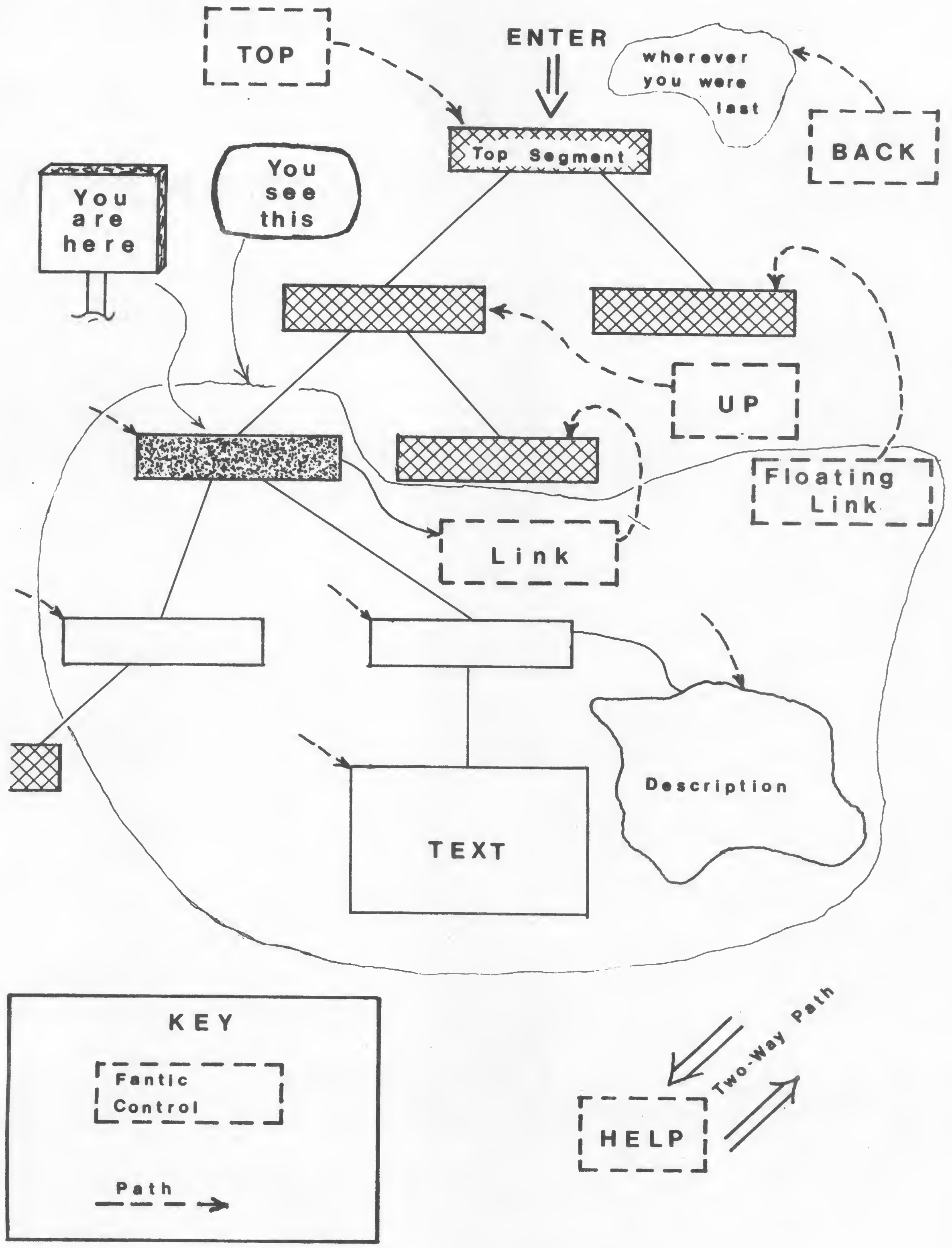


Figure 8: File Navigation

violated, the command is ignored. Note that the display depth is stacked along with the location during navigation and is thus restored by the Back command.

SEGMENT EDITING

Segment editing includes the formation of segments and the deletion and insertion of segments.

Four major verbs are used for creating segments. These are Cat, Repeat, IF and Text, which create the corresponding segment types. The appropriate object for each of these verbs is a structurable (undefined) segment.

Upon receiving a command to create a segment, WHIMSI queries which (structurable) segment is to be structured. The user may specify a segment or press Cancel. In the case that only one structurable segment appears on the screen, the query is redundant and thus is not made. Structuring proceeds immediately with the only available segment.

The exact sequence of events in creating a segment depends on the type of segment to be created. In each case one or more strings must be entered by the user. To facilitate this, the program automatically enters edit mode with an object string containing a request for the appropriate string.

In most cases (See below) one or more segment descriptions are required for the newly created subordinates. If the segment corresponding to a requested description is to be Null, the "description" entered should be the string *NULL* which causes creation of a Null segment instead of the Undefined segment created otherwise.

For a Text segment, the appropriate string is the text. A CAT segment requires two strings describing the two subordinate (presently undefined) segments.

Repeat and IF both require a special string prior to the subordinate descriptions. In the case of Repeat this is the repetition control clause; for IF it is the Boolean expression.

Upon completion of the required strings, the display returns to normal, with the new segment included.

In case the user completes a segment and subsequently discovers that it doesn't belong where it is (or anywhere, for that matter), he may delete it from its present location and optionally move it elsewhere.

Except when in edit mode, the Delete command may take as its object a segment. Since a segment deletion may strongly impact the structure, only a segment in the display may be deleted. If a link is used to specify it, a diagnostic will be generated and the operation refused. Likewise, the top or root segment may not be deleted. Upon specification of an acceptable segment for deletion, WHIMSI gives the user one final opportunity to Cancel the operation, requiring positive confirmation of the desire to delete.

Once an appropriate object for deletion is confirmed, the specified segment is removed from its position in the tree and replaced by an undefined segment. A floating link to the deleted segment is created and the system automatically enters edit mode to allow the user to name the link.

Once a segment has been deleted, it may be inserted at any structurable (undefined) segment. The command for this is Insert. The first of two objects for this command is the location for the insertion. However, if only one structurable segment is visible, the redundant query is not made and the dialog passes directly to the second object.

The second object is necessarily the segment to be inserted. It must be a segment which has previously been deleted and not inserted elsewhere. That is, it must not be the top (root) segment and it must not be subordinate to any segment. Consequently, it may be specified only by its link. If that link is deleted before the segment is inserted, the storage remains allocated but the segment is lost, unless another link to it exists.

LINK EDITING

Link editing includes the creation and deletion of links, floating links and segment chains.

Links and floating links are created by use of the Link command. The first object of this command is the segment to which the link is to be directed, and consequently the segment which may be referenced by pointing to it. This segment must not be Undefined or Null. The second object is the segment where the link is to be rooted. If this object is omitted (by pressing Back) a floating link is created.

Upon creation of a link or floating link, WHIMSI automatically enters the edit mode to allow the user to name the link. The name may be any string (up to 74 characters) which can be displayed on one line. Although it is possible to insert new-line characters, this is not recommended since the resulting lines (after the first) cannot be displayed.

A segment is added to a chain by the Chain command. Its first object is a link which references the segment after which the new segment is to be inserted. The second is the segment to be included. The link used for the first argument need not be part of a chain. If it is not, it will be formed into a chain without altering its other functions.

Since links, floating links and chain elements are treated uniformly as links after their creation, only a single command is required to delete any of them. The Delete command, which is also used to delete segments, may be used to delete a link. As is the case with segment deletion, WHIMSI first queries whether deletion is actually desired before completing the deletion. The user is given the option to Cancel.

TEXT EDITING

All entry and modification of strings is performed in edit mode. Edit mode may be entered from a full frame view of a string (described above) by selecting Edit, or by using Edit in the command mode and specifying the string. Note that the latter method is the only way to edit the name of a link. Edit mode is also entered automatically during some other functions, as explained above. Exit from edit mode is obtained by pressing Back or its equivalent, the End key on the alphanumeric keyboard.

Upon entering edit mode, a full frame view of the string is given, with a cursor appearing under the first character and the query: "Edit Mode: What To Do?"

Actions which may be performed in the edit mode include entering of text, insertion, deletion and interchanging of substrings, and repositioning of the cursor. Some of these actions may be performed in more than one way.

The simplest action is repositioning of the cursor. This may be done from the alphanumeric keyboard by means of the Advance, Back Space and Jump keys. However, when the distance to be moved is large, these keys may require excessive time for the move. Thus WHIMSI provides an alternative. Simply pointing at a character specifies repositioning of the cursor to that position. However, some minor adjustments may be required since the light pen is not completely accurate.

Entry of text may be done from the alphanumeric keyboard, wherever the cursor is placed. Text may be overstruck or added to at the end. However, the hardware requires a maximum string length and this may be reached for some long strings. In this case, the string may generally be extended by pressing the Insert key, which is described below.

Text may be inserted after any character by positioning the cursor to that character and pressing Insert. Since this command inserts a string (of up to one hundred) null characters after the indicated position and repositions the cursor to the first of them, the cursor seems to disappear. This is the normal behavior of the hardware and cannot be avoided conveniently. As the user types, characters appear and the text separates to make room for them. Upon any action on the programmed function keyboard or with the light pen, the insertion is terminated. However, the end of the string is always an insertion point.

Deletion of text may be done entirely at the alphanumeric keyboard by concurrent use of the Continue and Null keys. The overstriking of characters by the null character effectively deletes them from the string. However, this means may be cumbersome or (in some cases) impossible. Thus another option includes use of the Delete key of the programmed function keyboard. This key initiates a sequence in which the user is asked to delimit with the light pen the substring to be deleted.

Portions of the string may be interchanged by use of the INTCH command. The sequence initiated by this key asks the user to delimit the two substrings to be interchanged. Either of the two strings may be specified first, but they must not overlap. If they are not satisfactory, a diagnostic is generated and the user must start over. Caution should be taken in using this command, since the inaccuracy of the light pen might cause some unexpected results.

Although each string is treated internally as a unit, it may be formatted for display by using the NL (New Line) key. This key simply replaces the character over the cursor with the new-line symbol and advances the cursor. This may be used anywhere, including program text, to obtain the desired formatting. It has no effect on executability of the program produced if it is used only where a space is allowed. Note that this character does not appear on the alphanumeric keyboard and cannot be overwritten since the cursor may not be placed on it. Thus the only means to remove it is the Delete key. Note also that use of this key terminates any insertion which may be in progress.

OTHER FUNCTIONS

WHIMSI maintains a core image of that portion of the file which is currently being acted upon. All changes to the file are effected upon this core image, which is used from time to time to update the disk file.

Consequently, the user has the ability to control when actual file updates will occur, and to reverse recent changes before they are stored on the file.

Storing of changes on the file is a side effect of the navigation functions. Each time the display is moved by any mechanism to another segment, all changes (including string editing) made at the current location are made permanent. If the display is "moved" to the current segment, no actual movement takes place but the updates are stored on the file. This provides optional control of disk action.

Until the time updates are moved to disk, the user has the option of reversing them. The Restore key causes the core image to be refreshed from disk, reversing all changes since disk was last written. This provides one last chance to reverse a disastrous editing error.

<u>FILE NAVIGATION</u>

TOP -- Move to the root segment
UP -- Move to the parent segment
BACK -- Reverse the last move

Pen to Segment -- Move to the indicated segment
Pen to Link -- Move to the related segment
Pen to String -- Obtain full frame view of the string

<u>EDITING</u>

EDIT -- Enter Edit mode

INSERT -- Insert (characters or a segment)
DELETE -- Delete characters, or detatch a segment
INTCH -- Interchange two substrings

NL -- Generate the new-line symbol
BACK -- Exit from Edit mode

<u>STRUCTURING</u>

```
CAT -- Subdivide a segment
REPEAT --    "         "
IF --        "         "
TEXT --      "         "
```

LINK -- Create a (floating) link
CHAIN -- Add a segment to a chain

<u>MISCELLANEOUS</u>

HELP -- Enter the HELP mode

MORE DEPTH -- Increase or
LESS DEPTH -- Decrease the depth of display nesting

CANCEL -- Abort entry of a command
RESTORE -- Reverse recent changes to the file

Figure 9: Summary of WHIMSI Commands

Programming Methodology

Although the description of the controls given above is complete, there are some special methods for using them, which facilitate their use.

The Use of Subdivisions and Descriptions. Since WHIMSI requires a programmer to explicitly structure his program by successively subdividing the process, the approach may at first seem alien. However, a couple hours of practice, and the development of some appropriate habits, will make it seem more natural.

The first consideration is to avoid trying to imagine what the expanded PL/1 code will look like. The object of using WHIMSI is to obviate the use of linear text, mentally as well as visually. Some practice may be required, but it soon becomes convenient to treat each subdivision independently, subdividing it as required.

It is generally useful to consider each segment in terms of its initial and terminal environments. That is, certain relations hold between the program variables on entry to the segment, and that segment performs a function on those variables producing a new environment. Such a treatment not only facilitates discovery of the next appropriate subdivisions, but provides the basis for possible future verification (Elspas, Levitt, Waldinger and Waksman, 1972).

A description of the initial environment, together with the function to be performed, often shows a natural division. Such terms in the description as "and," "or" or "all" give heuristic evidence that the next subdivision may be CAT, IF or Repeat, respectively.

The use of segment descriptions can be made to facilitate this. For instance, let the first line (which is normally displayed) contain a short blurb giving a mnemonic description of the segment. Then let the remaining lines specify as much as may be needed the details of the initial and terminal environments, together with some description of the nature of the process. If this is done, the documentation is essentially complete.

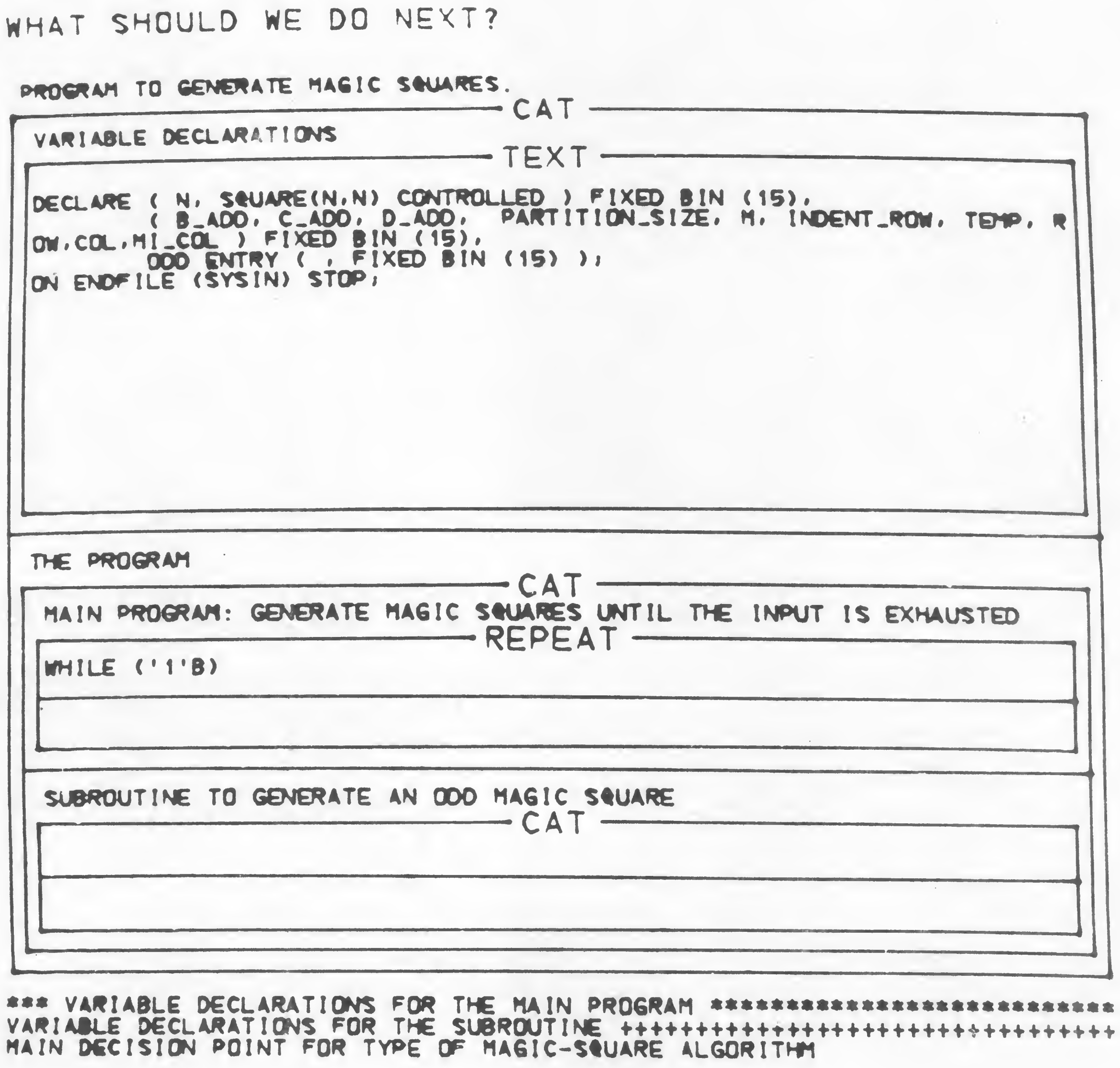
Declarations and Subroutines. Since this implementation of WHIMSI is required to deal with a linear-text object language (ie. PL/1) some small problems arise concerning the use of declarations and subroutines. Since each is properly associated with the point at which it is used, that would be the appropriate place to display it. Future versions will hopefully deal with this requirement, but this version requires special methods.

Declarations may be handled simply by following the convention that the first subdivision of the program is CAT, with the first subordinate segment containing all the variable declarations. The other subordinate thus comprises the executable portion of the program. Creation of a floating link to the declarations segment provides immediate access whenever required.

Subroutines may be handled in a similar manner. First subdivide the program segment by CAT. The second subordinate may then contain all the subroutines. Any number of subroutines may be accomodated by successive subdivision of the subroutines segment. It will generally prove useful to construct a link to a subroutine from each segment that references it.

Text formatting. Although the implementation of the NL key provides a means for formatting text into lines, some difficulties may arise from its use. During initial entry of text, its behavior is just as one would normally expect. However, if it is used in the midst of existing text during update, some unexpected results might be obtained.

As indicated above, use of NL terminates the insertion mode. Thus if it is used to insert more than one line, problems may arise. The simplest solution is to type multiple - line insertions as one line with an extra space between "lines" and then to overstrike the space with NL. Although this is somewhat inconvenient, it appears to be the best way to avoid difficulties.



```
WHAT SHOULD WE DO NEXT?

PROGRAM TO GENERATE MAGIC SQUARES.
                                CAT
VARIABLE DECLARATIONS
                                TEXT
DECLARE ( N, SQUARE(N,N) CONTROLLED ) FIXED BIN (15),
        ( B_ADD, C_ADD, D_ADD,  PARTITION_SIZE, M, INDENT_ROW, TEMP, R
OW,COL,MI_COL ) FIXED BIN (15),
         ODD ENTRY ( , FIXED BIN (15) );
ON ENDFILE (SYSIN) STOP;

THE PROGRAM
                                CAT
MAIN PROGRAM: GENERATE MAGIC SQUARES UNTIL THE INPUT IS EXHAUSTED
                                REPEAT
WHILE ('1'B)

SUBROUTINE TO GENERATE AN ODD MAGIC SQUARE
                                CAT

*** VARIABLE DECLARATIONS FOR THE MAIN PROGRAM ***************************
VARIABLE DECLARATIONS FOR THE SUBROUTINE +++++++++++++++++++++++++++++++++
MAIN DECISION POINT FOR TYPE OF MAGIC-SQUARE ALGORITHM
```

Figure 10: Declaration and Subroutine Divisions

Sometimes it may become necessary to delete an entire line of text. Again, the presence of the new-line symbol may create difficulties. The problem arises from the fact that, to delete a line, it is necessary to delete its trailing new-line symbol. Otherwise double spacing would occur. However, as noted above, the new-line symbol can be deleted only by use of the Delete key. This requires that visible characters delimit the string to be deleted.

Since a line is delimited by the invisible new-line character, a visible character must be placed after it to be deleted. Since the required position may be occupied by the first character of the next line, the result is less than desirable. However, if the convention is followed during entry that at least one space must precede a line end, a solution is possible.

To delete a line, place an arbitrary character just before the end of the preceding line, overstriking the final space. Then delete the string delimited by this new character and the end of the line to be deleted. The new-line character of the deleted line thus becomes part of that for the preceding line, replacing the one deleted.

Insertion prior to the first character of a line is another source of potential difficulty. Since insertion always occurs <u>after</u> the position of the cursor, this cannot be done directly. The best solution is to Insert after the first character, then Back Space and overstrike. This will require retyping of the original first character, which may be done as the last character of the insertion.

PART IV:

A SAMPLE

In order to further explain the use of WHIMSI, a sample dialog is included here, preceded by the program it produced. In order to save space, only the first two and the last two pages of the dialog are included.

The dialog is in the form of a log produced by the WHIMSI system. Each entry is preceded by the Julian date and the time at which it was made. Since it was originally designed for use during debugging and initial evaluation of the program, the log is in the form of a program trace.

Each entry of the log essentially records an event. Each exchange is thus recorded by a series of entries, beginning with one marked "* QUERY: " followed by a copy of the query presented to the user. In most cases, the first entry following the query indicates the device which the user used for his response. Subsequent entries indicate further analysis of the action and (in some cases) the action taken by the program.

Summary_of_the_Programming_Task. The task performed during the session recorded here was the creation of a program from specifications alone. That is, the function of the program was known, as was a general method for performing that function. However, there were no flow charts, notes or other preceding work.

The program generates a set of square matrices known as magic squares, according to a list of input integers. That is, for each input integer (N) the program produces an NxN magic square.

A magic square is a matrix containing the integers from 1 to N square, arranged so that all rows, columns and diagonals add to the same integer, namely N*(N**2+1)/2.

The general method is taken from notes prepared for a Computer Science programming class. The same programmer previously produced a shorter version of the program (with less error checking) in a period of about four hours. The present program, produced about a year later, required a few seconds more than two hours and ran correctly when first submitted.

Program and Programmer Performance. The time elapsed between the initiation of the system and the request by the programmer to execute the completed program was 2 hours and 25 seconds. During this time WHIMSI responded to 650 discrete actions by the programmer. Since actions on the alphanumeric keyboard are not directly accessible to the program, only those actions on the programmed function keyboard and with the light pen were recorded.

Since the time of each event was recorded, it was possible to calculate approximately how long each action required. However, since time is recorded only to the nearest second, the calculated times may differ somewhat from the actual times. This is particularly noticeable when the times are short. In particular, response times are not accurately calculated, which should be kept in mind while reading the following paragraphs.

Of the 650 responses, only one required more than one second as recorded. This occurred (at 4:39:06) in response to a request for segment display. Apparently this three-second delay was due to temporary overloading of the operating system, which delayed the necessary channel operations.

Another 47 responses recorded a delay of one second. The remainder did not require a measurable length of time.

Thus most of the time spent creating the program was spent by the user, typing text or thinking of what to do next. The maximum delay recorded between program action and subsequent user action was three minutes, fourteen seconds. The mean was 11.02 seconds.

It is worth noting that, during construction of the program, it was decided to modify its structure. Since two loops turned out to have comparable structures, it was decided to combine them into a single loop, catenating the inner loops. Since considerable work had been done below this level, the ability to rearrange segments as units was quite helpful.

The restructuring process began at 3:25:58 with the deletion of a segment. The process continued, intermingled with further construction of the program, until 4:16:43 when the last temporary link was deleted.

During this time, 294 discrete actions were taken, comprising 45 percent of the total work. Of these actions, 25 initiated structuring operations (other than insertion and deletion) which account for the majority of the user actions during the interval. Throughout this process, several segments remained detatched (deleted) from the program structure, awaiting their reinsertion into the appropriate locations.

The Record. The remaining pages of this part include the PL/1 source listing and subsequent output, followed by the log of the programming process.

```
        /* THIS IS A PL/1 SOURCE MODULE PRODUCED BY WHIMSI */                 | WHIMSI

STMT LEVEL NEST
                   /* THIS IS A PL/1 SOURCE MODULE PRODUCED BY WHIMSI */                        | WHIMSI
  1             MAGIC  : PROC OPTIONS (MAIN);                                                   | WHIMSI
                                                                                                | WHIMSI
                   /*  PROGRAM TO GENERATE MAGIC SQUARES.                                       | WHIMSI
                FOR EACH INPUT INTEGER (N) GENERATE AN NXN MAGIC SQUARE.    */                  | WHIMSI
                   /*  VARIABLE DECLARATIONS    */                                              | WHIMSI
  2     1       DECLARE ( N, SQUARE(N,N) CONTROLLED ) FIXED BIN (15),                           | WHIMSI
                      ( B_ADD, C_ADD, D_ADD,  PARTITION_SIZE, M, INDENT_ROW, TEMP, RO| WHIMSI
                W,COL,MI_COL ) FIXED BIN (15),                                                  | WHIMSI
                      ODD ENTRY ( , FIXED BIN (15) );                                           | WHIMSI
  3     1       ON ENDFILE (SYSIN) STOP;                                                        | WHIMSI
                   /*  THE PROGRAM    */                                                        | WHIMSI
                   /*  MAIN PROGRAM: GENERATE MAGIC SQUARES UNTIL THE INPUT IS EXHAUSTE | WHIMSI
                D   */                                                                          | WHIMSI
  5     1       DO WHILE ('1'B) ;                                                               | WHIMSI
                   /*  GENERATE A MAGIC SQUARE ACCORDING TO THE NEXT SPECIFICATION   */| WHIMSI
                                                                                                | WHIMSI
                   /*  OBTAIN THE SPECIFICATION (N)    */                                       | WHIMSI
  6     1    1  GET LIST ( N );                                                                 | WHIMSI
                   /*  GENERATE THE SQUARE IF THE SPECIFICATION IS ACCEPTABLE,                  | WHIMSI
                ELSE GENERATE A DIAGNOSTIC    */                                                | WHIMSI
  7     1    1   IF N > 2  &  N <= 40                                                           | WHIMSI
  8     1    1   THEN DO;                                                                       | WHIMSI
                                                                                                | WHIMSI
                   /*  GENERATE A MAGIC SQUARE(N,N)    */                                       | WHIMSI
                   /*  CREATE THE MAGIC SQUARE    */                                            | WHIMSI
                   /*  ALLOCATE AN ARRAY FOR THE MAGIC SQUARE    */                             | WHIMSI
  9     1    2  ALLOCATE SQUARE ( N, N );                                                       | WHIMSI
                   /*  ENTER THE NUMBERS IN THE ARRAY TO FORM A MAGIC SQUARE    */              | WHIMSI
 10     1    2   IF MOD ( N, 2 ) ¬= 0                                                           | WHIMSI
 11     1    2   THEN DO;                                                                       | WHIMSI
                                                                                                | WHIMSI
                   /*  CREATE AN "ODD" SQUARE    */                                             | WHIMSI
 12     1    3  CALL ODD ( SQUARE, N );                                                         | WHIMSI
 13     1    3   END; ELSE DO;                                                                  | WHIMSI
                                                                                                | WHIMSI
                   /*  CREATE AN "EVEN" SQUARE    */                                            | WHIMSI
 15     1    3   IF MOD ( N, 4 ) = 0                                                            | WHIMSI
 16     1    3   THEN DO;                                                                       | WHIMSI
                                                                                                | WHIMSI
                   /*  CREATE A "DOUBLY EVEN" SQUARE    */                                      | WHIMSI
                   /*  FIRST FILL THE SQUARE WITH SEQUENTIAL INTEGERS, BY ROWS    */            | WHIMSI
 17     1    4  DO I = 1 TO N ;                                                                 | WHIMSI
                   /*  ENTER THE I-TH ROW OF INTEGERS    */                                     | WHIMSI
 18     1    5  DO J = 1 TO N ;                                                                 | WHIMSI
                   /*  ENTER THE J-TH INTEGER OF THE I-TH ROW    */                             | WHIMSI
 19     1    6  SQUARE ( I,J )  =  N*(I-1) + J;                                                 | WHIMSI
 20     1    6   END;                                                                           | WHIMSI
 21     1    5   END;                                                                           | WHIMSI
                   /*  THEN EXCHANGE PAIRS OF ELEMENTS AS FOLLOWS:                              | WHIMSI
                BEGINNING WITH THE MAIN DIAGONAL, AND FOR EACH SUBSEQUENT FOURTH                | WHIMSI
                PARALLEL DIAGONAL, EXCHANGE MIRROR-IMAGE ELEMENTS.  DO THE SAME                 | WHIMSI
                FOR THE MIRROR-IMAGE DIAGONALS.    */                                           | WHIMSI
 22     1    4  DO J = 1 TO N BY 4 ;                                                            | WHIMSI
                   /*  PROCESS A DIAGONAL AND ITS MIRROR IMAGE    */                            | WHIMSI
```

```
          /* THIS IS A PL/1 SOURCE MODULE PRODUCED BY WHIMSI */              | WHIMSI

STMT LEVEL NEST

                       /*  INITIALIZE FOR THE DIAGONAL    */                                  | WHIMSI
 23     1     5     COL = J;  MI_COL = J + 3;                                                  | WHIMSI
                       /*  EXCHANGE ALONG THE DIAGONAL    */                                   | WHIMSI
 25     1     5     DO ROW = 1 TO N/2 WHILE ( COL <= N  |  MI_COL >= 1 ) ;                     | WHIMSI
                       /*  EXCHANGE A PAIR ALONG EACH DIAGONAL    */                           | WHIMSI
                       /*  EXCHANGE ALONG THE MAIN DIAGONAL    */                              | WHIMSI
 26     1     6      IF COL <= N                                                               | WHIMSI
 27     1     6      THEN DO;                                                                  | WHIMSI
                                                                                               | WHIMSI
                       /*  EXCHANGE A PAIR    */                                               | WHIMSI
 28     1     7     TEMP = SQUARE ( ROW, COL );                                                | WHIMSI
 29     1     7     SQUARE ( ROW, COL )  =  SQUARE ( N-ROW+1, N-COL+1 );                       | WHIMSI
 30     1     7     SQUARE ( N-ROW+1, N-COL+1 ) = TEMP;                                        | WHIMSI
 31     1     7      END; ELSE DO;                                                             | WHIMSI
                      /***  NOTHING  ***/                                                      | WHIMSI
 33     1     7      END;                                                                      | WHIMSI
                       /*  ECHANGE ALONG THE MIRROR-IMAGE DIAGONAL    */                       | WHIMSI
 34     1     6      IF MI_CCL >= 1                                                            | WHIMSI
 35     1     6      THEN DO;                                                                  | WHIMSI
                                                                                               | WHIMSI
                       /*  EXCHANGE A PAIR    */                                               | WHIMSI
 36     1     7     TEMP = SQUARE ( ROW, MI_COL );                                             | WHIMSI
 37     1     7     SQUARE ( ROW, MI_COL )  =  SQUARE ( N-ROW+1, N-MI_COL+1 );                 | WHIMSI
 38     1     7     SQUARE ( N-ROW+1, N-MI_COL+1 )  =  TEMP;                                   | WHIMSI
 39     1     7      END; ELSE DO;                                                             | WHIMSI
                      /***  NOTHING  ***/                                                      | WHIMSI
 41     1     7      END;                                                                      | WHIMSI
 42     1     6      END;                                                                      | WHIMSI
 43     1     5      END;                                                                      | WHIMSI
 44     1     4      END; ELSE DO;                                                             | WHIMSI
                                                                                               | WHIMSI
                       /*  CREATE A "SINGLY EVEN" SQUARE    */                                 | WHIMSI
                       /*  PARTITION THE SQUARE INTO FOUR "ODD" SQUARES AND FILL EACH PARTI    | WHIMSI
                    TION                                                                       | WHIMSI
                    WITH A MAGIC SQUARE, USING THE NUMBERS FROM 1 TO N**2 AS FOLLOWS:          | WHIMSI
                    CONSIDER THE PARTITION AS LABELED        A C                               | WHIMSI
                                                             D B                               | WHIMSI
                    NUMBERS ARE ASSIGNED AS IN AN "ODD" SQUARE, IN THE ORDER A,B,C,D.   */     | WHIMSI
                                                                                               | WHIMSI
                       /*  FILL THE UPPER-LEFT CORNER OR PARTITION    */                       | WHIMSI
 46     1     4     CALL ODD ( SQUARE, N/2 );                                                  | WHIMSI
                       /*  FILL THE REMAINING PARTITIONS, BASED ON THE FIRST    */             | WHIMSI
                       /*  INITIALIZE CONSTANTS FOR THE ENTRIES    */                          | WHIMSI
 47     1     4     B_ADD = (N/2)**2;                                                          | WHIMSI
 48     1     4     C_ADD = B_ADD * 2;                                                         | WHIMSI
 49     1     4     D_ADD = B_ADD * 3;                                                         | WHIMSI
 50     1     4     PARTITION_SIZE = N/2;                                                      | WHIMSI
                       /*  FILL THE REMAINING PARTITIONS, BASED ON THE FIRST    */             | WHIMSI
 51     1     4     DO I = 1 TO PARTITION_SIZE ;                                               | WHIMSI
                       /*  FILL THE I-TH ROW OF EACH REMAINING PARTITION    */                 | WHIMSI
 52     1     5     DO J = 1 TO PARTITION_SIZE ;                                               | WHIMSI
                       /*  FILL THE J-TH ELEMENT OF THE I-TH ROW IN EACH REMAINING PARTITIO    | WHIMSI
                    N   */                                                                     | WHIMSI
 53     1     6     SQUARE ( I+PARTITION_SIZE, J+PARTITION_SIZE ) = SQUARE (I,J) + B_ADD;      | WHIMSI
 54     1     6     SQUARE ( I,      J+PARTITION_SIZE ) = SQUARE (I,J) + C_ADD;                | WHIMSI
```

```
          /* THIS IS A PL/1 SOURCE MODULE PRODUCED BY WHIMSI */                | WHIMSI

STMT LEVEL NEST

 55     1    6     SQUARE ( I+PARTITION_SIZE, J      ) = SQUARE (I,J) + D_ADD;          | WHIMSI
 56     1    6      END;                                                                | WHIMSI
 57     1    5      END;                                                                | WHIMSI
                       /*  PERMUTE SOME ELEMENTS BETWEEN THE PARTITIONS   */             | WHIMSI
                       /*  INITIALIZE THE CONSTANTS USED FOR THE PERMUTATION   */        | WHIMSI
 58     1    4     M = ( N - 2 )  /  4;                                                 | WHIMSI
 59     1    4     INDENT_ROW = FLOOR ( PARTITION_SIZE/2 ) + 1;                         | WHIMSI
                       /*  PERFORM PERMUTATIONS BETWEEN PARTITIONS A,D AND C,B   */     | WHIMSI
 60     1    4     DO I = 1 TO PARTITION_SIZE ;                                         | WHIMSI
                       /*  PERMUTE SOME ELEMENTS BETWEEN PARTITIONS A,D AND C,B AS FOLLOWS: | WHIMSI
                                                                                        | WHIMSI
                   GIVEN M = (N-2)/4, INTERCHANGE THE FIRST M ELEMENTS OF CORRESPONDING  | WHIMSI
                   ROWS IN AND D, EXCEPT THE MIDDLE ROWS WHERE THE FIRST M ELEMENTS AFTER | WHIMSI
                                                                                        | WHIMSI
                   THE FIRST ARE EXCHANGED.                                             | WHIMSI
                   THEN EXCHANGE THE LAST M-   ELEMENTS OF CORRESPONDING ROWS IN C AND B. | WHIMSI
                    */                                                                  | WHIMSI
                       /*  PERMUTE ONE ROW BETWEEN PARTITIONS A AND D   */              | WHIMSI
 61     1    5     DO J = 1 TO M ;                                                      | WHIMSI
                       /*  PERMUTE ONE PAIR OF ELEMENTS BETWEEN A AND D   */            | WHIMSI
 62     1    6     COL = J + ( I = INDENT_ROW );                                        | WHIMSI
 63     1    6     TEMP = SQUARE ( I, COL );                                            | WHIMSI
 64     1    6     SQUARE ( I, COL )  =  SQUARE ( I+PARTITION_SIZE, COL ) ;             | WHIMSI
 65     1    6     SQUARE ( I+PARTITION_SIZE, COL ) = TEMP;                             | WHIMSI
 66     1    6      END;                                                                | WHIMSI
                       /*  PERMUTE A ROW BETWEEN PARTITIONS C AND B   */                | WHIMSI
 67     1    5     DO COL = N TO N-M+2 BY -1 ;                                          | WHIMSI
                       /*  EXCHANGE A PAIR OF ELEMENTS BETWEEN PARTITIONS C AND B   */  | WHIMSI
 68     1    6     TEMP = SQUARE ( I, COL );                                            | WHIMSI
 69     1    6     SQUARE ( I, COL )  =  SQUARE ( I+PARTITION_SIZE, COL );              | WHIMSI
 70     1    6     SQUARE ( I+PARTITION_SIZE, COL )  =  TEMP;                           | WHIMSI
 71     1    6      END;                                                                | WHIMSI
 72     1    5      END;                                                                | WHIMSI
 73     1    4      END;                                                                | WHIMSI
 74     1    3      END;                                                                | WHIMSI
                       /*  PRINT AND FREE THE MAGIC SQUARE   */                         | WHIMSI
                       /*  PRINT THE MAGIC SQUARE AT AN APPROPRIATE LOCATION ON THE PAGE | WHIMSI
                   */                                                                   | WHIMSI
                       /*  OBTAIN AN APPROPRIATE STARTING POSITION ON THE PAGE   */     | WHIMSI
                       /*  ASSURE SEPARATION FROM PREVIOUS SQUARE   */                  | WHIMSI
 75     1    2      IF LINENO (SYSPRINT) > 1                                            | WHIMSI
 76     1    2      THEN DO;                                                            | WHIMSI
                                                                                        | WHIMSI
                       /*  SPACE TO SEPARATE FROM THE PREVIOUS SQUARE   */              | WHIMSI
 77     1    3     PUT SKIP(5);                                                         | WHIMSI
 78     1    3      END; ELSE DO;                                                       | WHIMSI
                     /***  NOTHING  ***/                                                | WHIMSI
 80     1    3      END;                                                                | WHIMSI
                       /*  ASSURE THAT ADEQUATE SPACE REMAINS ON THE PAGE   */          | WHIMSI
 81     1    2      IF 60-LINENO(SYSPRINT) < 2*N+2                                      | WHIMSI
 82     1    2      THEN DO;                                                            | WHIMSI
                                                                                        | WHIMSI
                       /*  BEGIN ON A NEW PAGE   */                                     | WHIMSI
 83     1    3     PUT PAGE;                                                            | WHIMSI
 84     1    3      END; ELSE DO;                                                       | WHIMSI
```

```
              /* THIS IS A PL/1 SOURCE MODULE PRODUCED BY WHIMSI */               | WHIMSI

STMT LEVEL NEST

                       /***  NOTHING  ***/                                                      | WHIMSI
 86     1     3       END;                                                                      | WHIMSI
                         /*  PRINT THE SQUARE    */                                             | WHIMSI
 87     1     2      PUT SKIP DATA ( N );                                                       | WHIMSI
 88     1     2      PUT EDIT ( SQUARE ) ( SKIP(2),                                             | WHIMSI
                              (N) ( (N) F(4), SKIP(2) ) );                                      | WHIMSI
                         /*  FREE THE SQUARE    */                                              | WHIMSI
 89     1     2      FREE SQUARE;                                                               | WHIMSI
 90     1     2       END; ELSE DO;                                                             | WHIMSI
                                                                                                | WHIMSI
                         /*  GENERATE A DIAGNOSTIC     */                                       | WHIMSI
 92     1     2      PUT SKIP(5) EDIT ( ' N = ', N, ' IS NOT ACCEPTABLE')                       | WHIMSI
                                      ( A, F(4), A );                                           | WHIMSI
 93     1     2       END;                                                                      | WHIMSI
 94     1     1       END;                                                                      | WHIMSI
                         /*  SUBROUTINE TO GENERATE AN ODD MAGIC SQUARE    */                   | WHIMSI
                         /*  PROLOGUE FOR THE SUBROUTINE    */                                  | WHIMSI
 95     1            ODD: PROC ( SQUARE, DIM );                                                 | WHIMSI
 96     2            DECLARE ( SQUARE(*,*), DIM ) FIXED BIN (15),                               | WHIMSI
                                                                                                | WHIMSI
                          ( ENTRY, ROW,COL, NEXT_ROW,NEXT_COL, I,J ) FIXED BIN (15);            | WHIMSI
                         /*  REMAINDER OF THE SUBROUTINE    */                                  | WHIMSI
                         /*  BODY OF SUBROUTINE TO GENERATE AN ODD MAGIC SQUARE.                | WHIMSI
                     INPUT PARAMETER SQUARE(N,N) IS TO RECEIVE A MAGIC                          | WHIMSI
                     SQUARE IN THE UPPER LEFT CORNER, DIMENSIONED BY "DIM".                     | WHIMSI
                     "DIM" MUST BE ODD.    */                                                   | WHIMSI
 97     2             IF MOD ( DIM,2 ) = 1  &  DIM > 1                                          | WHIMSI
 98     2             THEN DO;                                                                  | WHIMSI
                                                                                                | WHIMSI
                         /*  "DIM" IS VALID -- GENERATE THE MAGIC SQUARE    */                  | WHIMSI
                         /*  INITIALIZE THE SQUARE    */                                        | WHIMSI
                         /*  ZERO THE SQUARE    */                                              | WHIMSI
 99     2     1      DO I = 1 TO DIM ;                                                          | WHIMSI
                         /*  ZERO ROW "I" OF THE SQUARE    */                                   | WHIMSI
100     2     2      DO J = 1 TO DIM ;                                                          | WHIMSI
                         /*  ZERO THE J-TH ELEMENT OF ROW "I" IN THE SQUARE    */               | WHIMSI
101     2     3      SQUARE ( I, J )  =  0;                                                     | WHIMSI
102     2     3       END;                                                                      | WHIMSI
103     2     2       END;                                                                      | WHIMSI
                         /*  LOCATE THE FIRST ENTRY    */                                       | WHIMSI
104     2     1      NEXT_ROW = 1;                                                              | WHIMSI
105     2     1      NEXT_COL = FLOOR ( DIM/2 ) + 1;                                            | WHIMSI
                         /*  GENERATE THE SQUARE'S ENTRIES    */                                | WHIMSI
106     2     1      DO ENTRY = 1 TO DIM**2 ;                                                   | WHIMSI
                         /*  PLACE "ENTRY" IN THE APPROPRIATE LOCATION AND SELECT THE NEXT LO   | WHIMSI
                     CATION    */                                                               | WHIMSI
                         /*  PLACE "ENTRY" IN THE PROPER LOCATION    */                         | WHIMSI
107     2     2       IF SQUARE ( NEXT_ROW, NEXT_COL ) = 0                                      | WHIMSI
108     2     2       THEN DO;                                                                  | WHIMSI
                                                                                                | WHIMSI
                         /*  MAKE THE ENTRY    */                                               | WHIMSI
109     2     3      SQUARE ( NEXT_ROW, NEXT_COL ) = ENTRY;                                     | WHIMSI
110     2     3       END; ELSE DO;                                                             | WHIMSI
                                                                                                | WHIMSI
                         /*  ERROR: SQUARE IS ALREADY OCCUPIED    */                            | WHIMSI
```

```
                 /* THIS IS A PL/1 SOURCE MODULE PRODUCED BY WHIMSI */              | WHIMSI

STMT LEVEL NEST

 112      2    3    PUT PAGE EDIT ('**** ERROR: LOCATION SELECTED FOR ENTRY IN "ODD"      | WHIMSI
                         IS ALREADY OCCUPIED.  ROW = ', NEXT_ROW, ', COLUMN = ',         | WHIMSI
                         NEXT_COL )   ( A,F(3),A,F(3) );                                 | WHIMSI
 113      2    3    PUT SKIP(3) LIST (SQUARE);                                            | WHIMSI
 114      2    3    STOP;                                                                 | WHIMSI
 115      2    3     END;                                                                 | WHIMSI
                       /*   SELECT THE NEXT LOCATION    */                                | WHIMSI
                       /*   MAKE A TENTATIVE SELECTION FOR THE NEXT LOCATION    */        | WHIMSI
 116      2    2    ROW = NEXT_ROW;  COL = NEXT_COL;                                      | WHIMSI
 118      2    2    NEXT_ROW = ROW - 1;                                                   | WHIMSI
 119      2    2    NEXT_COL = COL + 1;                                                   | WHIMSI
                       /*   ADJUST THE LOCATION IF NECESSARY    */                        | WHIMSI
                       /*   ASSURE THAT THE LOCATION IS WITHIN BOUNDS    */               | WHIMSI
 120      2    2     IF MAX ( DIM+1-NEXT_ROW, NEXT_COL )  >  DIM                          | WHIMSI
 121      2    2     THEN DO;                                                             | WHIMSI
                                                                                          | WHIMSI
                       /*   LOCATION IS OUT OF BOUNDS -- RESET    */                      | WHIMSI
                       /*   ASSURE THAT THE ROW IS WITHIN BOUNDS    */                    | WHIMSI
 122      2    3     IF NEXT_ROW < 1                                                      | WHIMSI
 123      2    3     THEN DO;                                                             | WHIMSI
                                                                                          | WHIMSI
                       /*   RESET ROW TO "DIM"    */                                      | WHIMSI
 124      2    4    NEXT_ROW = DIM;                                                       | WHIMSI
 125      2    4     END; ELSE DO;                                                        | WHIMSI
                      /***   NOTHING   ***/                                               | WHIMSI
 127      2    4     END;                                                                 | WHIMSI
                       /*   ASSURE THAT THE COLUMN IS WITHIN BOUNDS    */                 | WHIMSI
 128      2    3     IF NEXT_COL > DIM                                                    | WHIMSI
 129      2    3     THEN DO;                                                             | WHIMSI
                                                                                          | WHIMSI
                       /*   RESET COLUMN TO 1    */                                       | WHIMSI
 130      2    4    NEXT_COL = 1;                                                         | WHIMSI
 131      2    4     END; ELSE DO;                                                        | WHIMSI
                      /***   NOTHING   ***/                                               | WHIMSI
 133      2    4     END;                                                                 | WHIMSI
 134      2    3     END; ELSE DO;                                                        | WHIMSI
                      /***   NOTHING   ***/                                               | WHIMSI
 136      2    3     END;                                                                 | WHIMSI
                       /*   ASSURE THAT THE LOCATION IS NOT OCCUPIED    */                | WHIMSI
 137      2    2     IF SQUARE ( NEXT_ROW, NEXT_COL ) ¬= 0                                | WHIMSI
 138      2    2     THEN DO;                                                             | WHIMSI
                                                                                          | WHIMSI
                       /*   LOCATION IS OCCUPIED -- RESET TO ONE BELOW PREVIOUS LOCATION    *| WHIMSI
                    /                                                                     | WHIMSI
 139      2    3    NEXT_ROW = ROW + 1;                                                   | WHIMSI
 140      2    3    NEXT_COL = COL;                                                       | WHIMSI
 141      2    3     END; ELSE DO;                                                        | WHIMSI
                      /***   NOTHING   ***/                                               | WHIMSI
 143      2    3     END;                                                                 | WHIMSI
 144      2    2     END;                                                                 | WHIMSI
 145      2    1     END; ELSE DO;                                                        | WHIMSI
                                                                                          | WHIMSI
                       /*   "DIM" IS INVALID -- TERMINATE    */                           | WHIMSI
 147      2    1    PUT PAGE EDIT ('**** ERROR: SUBROUTINE "ODD" RECEIVED INVALID VALUE   | WHIMSI
                         DIM = ', DIM )   ( A, F(3) );                                   | WHIMSI
```

```
              /* THIS IS A PL/1 SOURCE MODULE PRODUCED BY WHIMSI */                    | WHIMSI
STMT LEVEL NEST

148     2    1    SIGNAL ERROR;                                                             | WHIMSI
149     2    1     END;                                                                     | WHIMSI
                     /*  END OF THE SUBROUTINE    */                                        | WHIMSI
150     2         END;                                                                      | WHIMSI

151     1         END;                                                                      | WHIMSI
```

```
N=          3:
    8   1   6
    3   5   7
    4   9   2


N=          4:
   16   2   3  13
   12   6   7   9
    8  10  11   5
    4  14  15   1


N=          7:
   30  39  48   1  10  19  28
   38  47   7   9  18  27  29
   46   6   8  17  26  35  37
    5  14  16  25  34  36  45
   13  15  24  33  42  44   4
   21  23  32  41  43   3  12
   22  31  40  49   2  11  20
```

```
N=        10;
  92  99   1   8  15  67  74  51  58  40
  98  80   7  14  16  73  55  57  64  41
   4  81  88  20  22  54  56  63  70  47
  85  87  19  21   3  60  62  69  71  28
  86  93  25   2   9  61  68  75  52  34
  17  24  76  83  90  42  49  26  33  65
  23   5  82  89  91  48  30  32  39  66
  79   6  13  95  97  29  31  38  45  72
  10  12  94  96  78  35  37  44  46  53
  11  18 100  77  84  36  43  50  27  59


N=        13;
  93 108 123 138 153 168   1  16  31  46  61  76  91
 107 122 137 152 167  13  15  30  45  60  75  90  92
 121 136 151 166  12  14  29  44  59  74  89 104 106
 135 150 165  11  26  28  43  58  73  88 103 105 120
 149 164  10  25  27  42  57  72  87 102 117 119 134
 163   9  24  39  41  56  71  86 101 116 118 133 148
   8  23  38  40  55  70  85 100 115 130 132 147 162
  22  37  52  54  69  84  99 114 129 131 146 161   7
  36  51  53  68  83  98 113 128 143 145 160   6  21
  50  65  67  82  97 112 127 142 144 159   5  20  35
  64  66  81  96 111 126 141 156 158   4  19  34  49
  78  80  95 110 125 140 155 157   3  18  33  48  63
  79  94 109 124 139 154 169   2  17  32  47  62  77
```

```
75.205   2:41:58   WHIMSI IS UP
75.205   2:41:58   * QUERY: 'WHAT WOULD YOU LIKE TO SEE?'
75.205   2:42:10   PEN DETECT
75.205   2:42:10   PROGRAM FILE SELECTED
75.205   2:42:11   * QUERY: 'WHAT SHOULD WE DO NEXT?'
75.205   2:42:16   KEY PRESS
75.205   2:42:16   EDIT SELECTED
75.205   2:42:16   * QUERY: 'WHICH TEXT DO YOU WANT TO EDIT?'
75.205   2:42:17   PEN DETECT
75.205   2:42:17   * QUERY: 'EDIT MODE: WHAT TO DO?'
75.205   2:42:28   KEY PRESS
75.205   2:42:28   NEW-LINE
75.205   2:42:28   * QUERY: 'WHAT NEXT?'
75.205   2:42:46   KEY PRESS
75.205   2:42:46   INSERT
75.205   2:42:46   * QUERY: 'WHAT NEXT?'
75.205   2:42:56   "END" KEY INTERPRETED AS "BACK"
75.205   2:42:56   END OF EDIT MODE
75.205   2:42:56   * QUERY: 'WHAT SHOULD WE DO NEXT?'
75.205   2:43:01   KEY PRESS
75.205   2:43:01   STRUCTURING COMMAND RECEIVED
75.205   2:43:01   SEGMENT TO BE STRUCTURED IDENTIFIED BY CONTEXT
75.205   2:43:01   * QUERY: 'EDIT MODE: WHAT TO DO?'
75.205   2:43:13   "END" KEY INTERPRETED AS "BACK"
75.205   2:43:13   END OF EDIT MODE
75.205   2:43:13   * QUERY: 'EDIT MODE: WHAT TO DO?'
75.205   2:43:23   "END" KEY INTERPRETED AS "BACK"
75.205   2:43:23   END OF EDIT MODE
75.205   2:43:23   STRUCTURING OPERATION COMPLETED
75.205   2:43:23   * QUERY: 'WHAT SHOULD WE DO NEXT?'
75.205   2:43:31   KEY PRESS
75.205   2:43:31   STRUCTURING COMMAND RECEIVED
75.205   2:43:31   * QUERY: '"CATENATE": WHICH SEGMENT?'
75.205   2:43:32   PEN DETECT
75.205   2:43:32   * QUERY: 'EDIT MODE: WHAT TO DO?'
75.205   2:44:02   "END" KEY INTERPRETED AS "BACK"
75.205   2:44:02   END OF EDIT MODE
75.205   2:44:02   * QUERY: 'EDIT MODE: WHAT TO DO?'
75.205   2:44:18   "END" KEY INTERPRETED AS "BACK"
75.205   2:44:18   END OF EDIT MODE
75.205   2:44:18   STRUCTURING OPERATION COMPLETED
75.205   2:44:18   * QUERY: 'WHAT SHOULD WE DO NEXT?'
75.205   2:44:24   KEY PRESS
75.205   2:44:24   STRUCTURING COMMAND RECEIVED
75.205   2:44:24   * QUERY: '"TEXT": WHICH SEGMENT?'
75.205   2:44:26   PEN DETECT
75.205   2:44:26   * QUERY: 'EDIT MODE: WHAT TO DO?'
75.205   2:44:49   KEY PRESS
75.205   2:44:49   NEW-LINE
75.205   2:44:49   * QUERY: 'WHAT NEXT?'
75.205   2:45:02   KEY PRESS
75.205   2:45:02   NEW-LINE
75.205   2:45:02   * QUERY: 'WHAT NEXT?'
75.205   2:45:06   KEY PRESS
75.205   2:45:06   INSERT
75.205   2:45:06   * QUERY: 'WHAT NEXT?'
75.205   2:45:14   "END" KEY INTERPRETED AS "BACK"
75.205   2:45:14   END OF EDIT MODE
75.205   2:45:14   STRUCTURING OPERATION COMPLETED
75.205   2:45:14   * QUERY: 'WHAT SHOULD WE DO NEXT?'
75.205   2:45:16   KEY PRESS
75.205   2:45:16   LINK SELECTED
```

```
75.205  2:45:16  * QUERY: 'LINK: TO WHICH SEGMENT?'
75.205  2:45:18  PEN DETECT
75.205  2:45:18  * QUERY: '... FROM WHERE?     ("BACK" GIVES FLOATING LINK)'
75.205  2:45:20  "END" KEY INTERPRETED AS "BACK"
75.205  2:45:20  FLOATING LINK INSERTED
75.205  2:45:20  ENTERING EDIT MODE TO NAME THE LINK
75.205  2:45:20  * QUERY: 'EDIT MODE: WHAT TO DO?'
75.205  2:45:34  "END" KEY INTERPRETED AS "BACK"
75.205  2:45:34  END OF EDIT MODE
75.205  2:45:34  LINK COMPLETED
75.205  2:45:34  * QUERY: 'WHAT SHOULD WE DO NEXT?'
75.205  2:45:50  PEN DETECT
75.205  2:45:50  DISPLAY MOVED TO INDICATED SEGMENT
75.205  2:45:50  * QUERY: 'WHAT SHOULD WE DO NEXT?'
75.205  2:45:55  KEY PRESS
75.205  2:45:55  STRUCTURING COMMAND RECEIVED
75.205  2:45:55  SEGMENT TO BE STRUCTURED IDENTIFIED BY CONTEXT
75.205  2:45:55  * QUERY: 'EDIT MODE: WHAT TO DO?'
75.205  2:46:15  "END" KEY INTERPRETED AS "BACK"
75.205  2:46:15  END OF EDIT MODE
75.205  2:46:15  * QUERY: 'EDIT MODE: WHAT TO DO?'
75.205  2:46:26  "END" KEY INTERPRETED AS "BACK"
75.205  2:46:26  END OF EDIT MODE
75.205  2:46:26  STRUCTURING OPERATION COMPLETED
75.205  2:46:26  * QUERY: 'WHAT SHOULD WE DO NEXT?'
75.205  2:46:29  KEY PRESS
75.205  2:46:29  STRUCTURING COMMAND RECEIVED
75.205  2:46:29  * QUERY: '"CATENATE": WHICH SEGMENT?'
75.205  2:46:30  PEN DETECT
75.205  2:46:30  * QUERY: 'EDIT MODE: WHAT TO DO?'
75.205  2:46:49  KEY PRESS
75.205  2:46:49  NEW-LINE
75.205  2:46:49  * QUERY: 'WHAT NEXT?'
75.205  2:47:06  KEY PRESS
75.205  2:47:06  INSERT
75.205  2:47:06  * QUERY: 'WHAT NEXT?'
75.205  2:47:24  KEY PRESS
75.205  2:47:24  NEW-LINE
75.205  2:47:24  * QUERY: 'WHAT NEXT?'
75.205  2:47:41  KEY PRESS
75.205  2:47:41  NEW-LINE
75.205  2:47:41  * QUERY: 'WHAT NEXT?'
75.205  2:47:48  KEY PRESS
75.205  2:47:48  INSERT
75.205  2:47:48  * QUERY: 'WHAT NEXT?'
75.205  2:47:52  "END" KEY INTERPRETED AS "BACK"
75.205  2:47:52  END OF EDIT MODE
75.205  2:47:52  * QUERY: 'EDIT MODE: WHAT TO DO?'
75.205  2:47:59  "END" KEY INTERPRETED AS "BACK"
75.205  2:47:59  END OF EDIT MODE
75.205  2:47:59  STRUCTURING OPERATION COMPLETED
75.205  2:47:59  * QUERY: 'WHAT SHOULD WE DO NEXT?'
75.205  2:48:01  KEY PRESS
75.205  2:48:01  STRUCTURING COMMAND RECEIVED
75.205  2:48:01  * QUERY: '"TEXT": WHICH SEGMENT?'
75.205  2:48:02  PEN DETECT
75.205  2:48:02  * QUERY: 'EDIT MODE: WHAT TO DO?'
75.205  2:48:08  "END" KEY INTERPRETED AS "BACK"
75.205  2:48:08  END OF EDIT MODE
75.205  2:48:08  STRUCTURING OPERATION COMPLETED
75.205  2:48:08  * QUERY: 'WHAT SHOULD WE DO NEXT?'
75.205  2:48:24  KEY PRESS
```

```
75.205   4:40:30   KEY PRESS
75.205   4:40:30   LINK SELECTED
75.205   4:40:30   * QUERY: 'LINK: TO WHICH SEGMENT?'
75.205   4:40:32   PEN DETECT
75.205   4:40:32   * QUERY: '... FROM WHERE?     ("BACK" GIVES FLOATING LINK)'
75.205   4:40:35   KEY PRESS
75.205   4:40:35   FLOATING LINK INSERTED
75.205   4:40:35   ENTERING EDIT MODE TO NAME THE LINK
75.205   4:40:35   * QUERY: 'EDIT MODE: WHAT TO DO?'
75.205   4:40:36   KEY PRESS
75.205   4:40:36   END OF EDIT MODE
75.205   4:40:36   LINK COMPLETED
75.205   4:40:36   * QUERY: 'WHAT SHOULD WE DO NEXT?'
75.205   4:40:40   KEY PRESS
75.205   4:40:41   DISPLAY RESET TO TOP OF FILE
75.205   4:40:41   * QUERY: 'WHAT SHOULD WE DO NEXT?'
75.205   4:40:47   PEN DETECT
75.205   4:40:47   LINK TO SEGMENT:
75.205   4:40:47   MAIN DECISION POINT FOR TYPE OF MAGIC-SQUARE ALGORITHM
75.205   4:40:47   DISPLAY MOVED TO INDICATED SEGMENT
75.205   4:40:48   * QUERY: 'WHAT SHOULD WE DO NEXT?'
75.205   4:40:51   KEY PRESS
75.205   4:40:51   LINK SELECTED
75.205   4:40:51   * QUERY: 'LINK: TO WHICH SEGMENT?'
75.205   4:40:55   PEN DETECT
75.205   4:40:55   LINK TO SEGMENT:
75.205   4:40:55   *** REPLACE THIS STRING WITH THE FLOATING LINK'S NAME ***
75.205   4:40:55   * QUERY: '... FROM WHERE?     ("BACK" GIVES FLOATING LINK)'
75.205   4:41:01   PEN DETECT
75.205   4:41:01   STANDARD LINK INSERTED
75.205   4:41:01   ENTERING EDIT MODE TO NAME THE LINK
75.205   4:41:01   * QUERY: 'EDIT MODE: WHAT TO DO?'
75.205   4:41:13   "END" KEY INTERPRETED AS "BACK"
75.205   4:41:13   END OF EDIT MODE
75.205   4:41:13   LINK COMPLETED
75.205   4:41:13   * QUERY: 'WHAT SHOULD WE DO NEXT?'
75.205   4:41:17   PEN DETECT
75.205   4:41:17   DISPLAY MOVED TO INDICATED SEGMENT
75.205   4:41:18   * QUERY: 'WHAT SHOULD WE DO NEXT?'
75.205   4:41:21   KEY PRESS
75.205   4:41:22   "BACK" -- RETRACE PATH
75.205   4:41:22   * QUERY: 'WHAT SHOULD WE DO NEXT?'
75.205   4:41:37   PEN DETECT
75.205   4:41:38   DISPLAY MOVED TO INDICATED SEGMENT
75.205   4:41:38   * QUERY: 'WHAT SHOULD WE DO NEXT?'
75.205   4:41:44   KEY PRESS
75.205   4:41:44   LINK SELECTED
75.205   4:41:44   * QUERY: 'LINK: TO WHICH SEGMENT?'
75.205   4:41:48   PEN DETECT
75.205   4:41:48   LINK TO SEGMENT:
75.205   4:41:48   *** REPLACE THIS STRING WITH THE FLOATING LINK'S NAME ***
75.205   4:41:48   * QUERY: '... FROM WHERE?     ("BACK" GIVES FLOATING LINK)'
75.205   4:41:52   PEN DETECT
75.205   4:41:52   STANDARD LINK INSERTED
75.205   4:41:52   ENTERING EDIT MODE TO NAME THE LINK
75.205   4:41:52   * QUERY: 'EDIT MODE: WHAT TO DO?'
75.205   4:42:01   "END" KEY INTERPRETED AS "BACK"
75.205   4:42:01   END OF EDIT MODE
75.205   4:42:01   LINK COMPLETED
75.205   4:42:02   * QUERY: 'WHAT SHOULD WE DO NEXT?'
75.205   4:42:05   KEY PRESS
75.205   4:42:05   * QUERY: 'DELETE: WHICH SEGMENT OR LINK?'
```

```
   75.205  4:42:07  PEN DETECT
   75.205  4:42:07  * QUERY: 'PRESS "BACK" TO DELETE THE LINK; ELSE "CANCEL"'
   75.205  4:42:09  KEY PRESS
   75.205  4:42:09  LINK DELETED
   75.205  4:42:09  * QUERY: 'WHAT SHOULD WE DO NEXT?'
   75.205  4:42:21  KEY PRESS
   75.205  4:42:21   "HELP" REQUESTED
   75.205  4:42:21  * QUERY: 'WHAT WOULD YOU LIKE TO SEE?'
   75.205  4:42:23  PEN DETECT
   75.205  4:42:23  EXIT SELECTED
   75.205  4:42:24  PEN DETECT
   75.205  4:42:24  "EXECUTE" SELECTED


WHIMSI   -- JOB 610 -- HASP-II OUTPUT COMPLETE AT  4:56
```

CONCLUSIONS

Experimentation with WHIMSI has yielded a wealth of information concerning the workability and requirements of such a system for programming. This information will motivate some different approaches to the next implementation, as well as some further research concerning methods of program design.

Since the principal purpose of this implementation was to investigate the strengths and weaknesses of the concept, such are the topics of these conclusions.

Although they are outweighed by the successes, the shortcomings are discussed first. Some problems arise from the inflexibility of the file structure. Others arise from the structure of the dialog and from difficulties with the display conventions.

These shortcomings lead to a discussion of the successes, which are sufficiently encouraging to warrant further experimentation with the WHIMSI concept.

Room For Improvement. Although it proved sufficient for the development of relatively small programs (See the Sample) the file structure is not sufficiently flexible to facilitate larger programs.

The first difficulty with the structure is that it allows only two elements in a catenation of segments. When the design decisions were made, this did not appear to be serious since any number of segments can be effectively catenated by successive subdivision. However, this forced method results in unnatural divisions and artificially deep nesting of the displayed structure.

Another shortcoming of the structure is the absence of a segment type analogous to the _case_ statement of Algol. Although it can be simulated to some extent by nesting of IFs, this is not adequate. PL/1 provides for simulation of _case_ by means of an array of statement labels, but this capability is effectively removed by WHIMSI since it discourages the use of statement labels. Nevertheless, this shortcoming is not intrinsic to the overall design, and could be remedied by a simple modification of the program.

Another shortcoming is actually due to an excess. The segment chaining facility does not appear to be very useful for programming. Although its use for keeping track of Undefined segments is essential, this job can be better done another way. Indeed, it was done this way only because the chain was to be available and promised to simplify the facility. However, a special - purpose list together with a menu might prove to be much better since it would inform the user immediately how many segments remain undefined, and of what they are. Furthermore, it would allow navigation to any Undefined segment in a single step without passing over a (possibly long) chain.

The difficulties mentioned above can be remedied by simple modifications of WHIMSI, but another remains which will require complete redesign of the file structure. The problem arises mainly from the fact that each segment is allowed to be subordinate to not more than one other segment, and that only one segment (ie. the root) defines a program.

As a result of this design restriction, segments may not share their subordinates. For small programs this is generally not a problem, but larger programs tend to include many parts which are similar if not identical. This tendency is usually reflected in a large number of macro references and/or subroutine calls. However, in the current implementation of WHIMSI, inclusion of subroutines is clumsy at best.

Even if this restriction were removed, another problem would remain. Sharing of segments between programs would remain impossible, but it is just such a facility which could make WHIMSI truly useful for programming. Many programs include segments which are similar to those in others. If those other segments were available during construction of a new program, considerably less work would be required and less errors would occur.

Although it would be a simple matter to allow multiple segments to reference a single segment, other problems could arise. For instance, if unlimited license were granted to subordinate one segment to another, a segment might inadvertantly become subordinate to itself. This situation would greatly complicate the algorithms for expansion of the code, since this condition must be detected. The best time to prevent such loops would be during initial construction of the program, and a method for doing this is under investigation.

Reference to segments by location (as is the case in the current implementation) leads to other difficulties. If it becomes necessary to physically move a segment, it is not convenient to update all references to it. In the currently limited environment this can be bad enough, but in a large program base with many references, the problem could be insurmountable. Thus it appears better to refer to segments by name, mediated by a hash table.

In addition to the shortcomings of the file structure, the structure of the dialog leaves a bit to be desired.

The most noticeable difficulty with the dialog is due to the inaccuracy of the light pen. Especially during text editing operations, unexpected results are sometimes obtained, thus deterring extensive use of some options. This situation can be remedied either by improving light pen accuracy through improved feedback (the preferred solution) or by substituting a more accurate device such as a mouse.

During the specification of a segment, the dialog takes the user to the edit mode. Since this occasionally includes the creation of text, another problem arises. The edit mode removes from view all information apart from the string being edited. Thus if a programmer needs to reference (for instance) the declaration of a variable to be used it is necessary to leave edit mode to do so. This is particularly bothersome during specification of descriptions since exit from edit mode generally enters edit mode for another description.

This could be effectively remedied by allowing the user to suspend an operation to perform another. This is indeed provided by the HELP command, and could be generalized to any operation. However, this would require further restructuring of the dialog since it might leave some operations in a partially completed state, possibly affecting the validity of the intervening operations. A model for an appropriate dialog structure is presented in a previous unpublished paper.

Another apparent shortcoming of the dialog structure is revealed by the tendency of some programmers to code short IF statements within a text segment rather than use the built-in facility. Since this version of WHIMSI does no syntax checking and simply expands the text, this action is accepted. However, future versions are intended to check syntax and immediately generate machine code, which would invalidate the action.

More important however is the fact that this tendency reveals a weakness in the overall design of the dialog. The object is to make things easier by doing them the new way.

This seems to indicate that, at least for short IF statements, the old way is easier. The major reason appears to be the requirement to provide an abstract description for each clause, each of which may be only a single statement. Perhaps the dialog should be restructured to allow some relaxation of this requirement and to further simplify entering of program segments.

As mentioned above, WHIMSI performs no syntax checking on the input text. This requires the user to rely on the off-line syntax checking performed by the compiler. This shortcoming will of course be remedied by inclusion of compilation on-line.

In addition to the required changes mentioned above, some modifications of the display algorithms would be in order.

On some occasions, the screen becomes terribly cluttered. In particular, it becomes difficult on occasion to differentiate between nested and catenated segments. This generally occurs when catenations are nested within catenations. This in turn occurs when the user actually intends to catenate more than two or three segments. Thus the problem can be alleviated to a great extent by inclusion of a multiple - element catenation segment.

Another oversight, which reduces the effectiveness of the links, is the limitation of link display to those associated with the "current" segment. It would be more useful if all links associated with visible segments were displayed. It may thus become necessary to devise a means of indicating which of the links displayed belongs to which segment.

Finally, it would be useful on occasion if the screen could be split, displaying simultaneously information from two (or possibly more) portions of the structure. Such a feature would be a natural complement to the proposed capability to suspend an operation.

Successes. The principal successes of the WHIMSI experiment are summarized above. That is, much has been learned about how to make the system better when it is completed.

During the original design phase, it was difficult if not impossible to anticipate how certain features would work in practice. There is not sufficient data in the literature to predict in detail the manner in which such a medium as WHIMSI would be used.

Experimentation with the system, limited as it was by its compromises in design, yielded information about some unsuspected requirements and suggested different approaches to others. This will save much effort in the design of subsequent versions.

However, all this information would be virtually worthless if there were no evidence that the WHIMSI concept itself is a good one. Fortunately, such evidence has been obtained.

The mere act of constructing several non-trivial programs using WHIMSI proves that the fantic controls are essentially sufficient for program construction. Furthermore, the speed with which programs can be written using the system indicates that the fantic controls are essentially sufficient and that the fantic space is conceptually useful.

Optimism is further accentuated by the fact that the observed benefits occurred in spite of the obvious limitations of the experimental version. Since most shortcomings are easily corrected (and none seem insurmountable) it is possible that far greater benefits will be obtained with subsequent versions.

However, this optimism must be somewhat tempered by the fact that the benefits have not been verified by a truly controlled experiment. Convincing though the evidence may be at first sight, it nevertheless leaves some doubts.

To date, the total user population for WHIMSI is two graduate students, one of which has difficulties due to incomplete knowledge of the PL/1 syntax. The other is the system's designer. This does not represent a valid sample of the potential user population and thus some doubts must remain concerning the validity of the observations. Nevertheless, the observed benefits do constitute an existence proof that the system is beneficial to some programmers.

Experimentation with larger samples has been prevented partly by lack of available time, and partly by the intrinsic difficulties of designing a meaningful test. Previous experiments aimed at measuring the relative merits of competing systems (Sackman, 1970) have failed.

Sackman experimented with groups of programmers, measuring only aggregate results. Consequently the observed "differences" in effectiveness were not statistically significant. Variance between programmers exceeded that between groups. Nothing can be said from these experiments about the benefits to individual programmers which might arise from use of a particular system.

Experimentation with system effects on individuals can be equally difficult if not worse. If a programmer is to be tested on two competing systems, precautions must be taken to avoid bias. It is extremely difficult to control for the difficulty of the programming task or for the quality of the program produced. Since neither of these quantities can be operationally defined without resort to opinion, any definition runs the risk of being contaminated by prejudice.

Furthermore, the subject's own bias may affect the results. When asked to alternate between two systems, a programmer is likely to develop a preference. Unless this preference (however rational or irrational it may be) is to be considered part of the evaluation of the systems, it may unduly influence the programmer's performance.

All these difficulties appear to be intrinsic to the attempt to evaluate entire systems. Such an attempt is basically incorrect anyway, since the many features of a system interact to produce the overall result. Observable differences between systems might be due to a single feature in one which could just as well be added to the other. On the other hand, a single feature may be enough to mask differences which would otherwise be clear.

Consequently, it appears that the best approach to testing the psychological effectiveness of design concepts is not to test systems but rather features. Experiments of this sort are difficult to design, but some methodology is being developed (Shneiderman, 1975; Sime, Green and Guest, 1973). Results remain primitive, and so the best available tool for evaluating systems appears to be an intuitive judgement based on an evaluation in the light of available evidence.

In the light of available evidence, which motivated the initial design of WHIMSI, the concept appears to have merit.

APPENDIX I:

JCL AND OPERATING REQUIREMENTS

This appendix specifies the JCL necessary to use WHIMSI under the OS/360 operating system. Also included are three additional utility programs which facilitate use of the system.

<u>The Main System.</u> The main system includes the graphics support program (WHIMSI) for entering and updating programs, and another program (EXPAND) which translates the structured file into a PL/1 program. Additional programs used are the PL/1 compiler and the system's program loader. These are invoked by a procedure (See Figure 11) which provides the total services explained in Part III.

Upon termination WHIMSI sets the condition code to indicate which, if any, of the other programs in the procedure are to be executed. If execution is selected, the condition code is set to zero, permitting all steps except EXP to execute. The latter step is similar to the EXPAND step, but prints the PL/1 text instead of passing it to the compiler. Thus its use is mutually exclusive with the other, to be used when expansion is desired without compilation. A condition code of four allows execution of the compiler but not of the compiled program, and a code of eight suppresses compilation, allowing the PL/1 text to be printed. If the condition code is set to 12, all further steps are suppressed. These options correspond to the items on the exit menu (Figure 12).

```
//WHIMCG PROC UNIT=3330,VOL=SYS301,DSN=PROGRAM,NAME=TEST
//WHIM    EXEC PGM=WHIMSI,REGION=(,110K)
//STEPLIB   DD UNIT=3330,VOL=SER=SYS301,DISP=SHR,
//             DSN=WHIMSI.LIBRARY
//SYSUDUMP DD SYSOUT=A
//G2250     DD UNIT=2250-1,DISP=OLD
//MANUAL    DD UNIT=3330,VOL=SER=SYS301,DISP=OLD,DSN=MANUAL
//PROGRAM   DD UNIT=&UNIT,VOL=SER=&VOL,DISP=OLD,DSN=&DSN
//LOG       DD SYSOUT=A
//EXPAND EXEC PGM=EXPAND,TIME=(,10),REGION=50K,
//             PARM=&NAME,COND=(8,LE,WHIM)
//STEPLIB DD   UNIT=2314,VOL=SER=SYS006,DISP=SHR,
//             DSN=WHIMSI.LIBRARY
//SYSPRINT DD SYSOUT=A
//PL1OUT    DD UNIT=SYSDA,DISP=(,PASS),DSN=&PL1,
//             DCB=BLKSIZE=3120,SPACE=(CYL,(1,1))
//PROGRAM   DD UNIT=&UNIT,VOL=SER=&VOL,DISP=OLD,DSN=&DSN
//SYSUDUMP DD SYSOUT=A
//EXP     EXEC PGM=EXPAND,TIME=(,10),REGION=50K,
//             PARM=&NAME,COND=(8,NE,WHIM)
//STEPLIB DD   UNIT=2314,VOL=SER=SYS006,DISP=SHR,
//             DSN=WHIMSI.LIBRARY
//SYSPRINT DD SYSOUT=A
//PL1OUT    DD SYSOUT=A,DCB=BLKSIZE=80
//PROGRAM   DD UNIT=&UNIT,VOL=SER=&VOL,DISP=OLD,DSN=&DSN
//SYSUDUMP DD SYSOUT=A
//PL1L   EXEC  PGM=IEMAA,REGION=100K,
//             COND=(8,LE,WHIM),PARM='ATR,XREF'
//SYSPRINT DD SYSOUT=A,DCB=(LRECL=125,BLKSIZE=129,RECFM=VBA)
//SYSLIN    DD DSN=&&LOADSET,UNIT=SYSDA,
//             SPACE=(960,(120,120)),DISP=(NEW,PASS),
//             DCB=(LRECL=80,BLKSIZE=960,RECFM=FB)
//SYSUT3    DD DSN=&&SYSUT3,UNIT=SYSDA,SPACE=(1600,(240,80))
//SYSUT1    DD DSN=&&SYSUT1,UNIT=SYSDA,SPACE=(1600,(240,80))
//SYSIN     DD UNIT=SYSDA,DSN=&PL1,DISP=(OLD,DELETE)
//GO   EXEC    PGM=LOADER,PARM='MAP',REGION=100K,
//             COND=((0,NE,WHIM),(4,LT,EXPAND),(4,LT,PL1L))
//SYSLIB    DD DSNAME=SYS1.PL1LIB,DISP=SHR
//          DD DSNAME=SYS1.USERLIB,DISP=SHR
//SYSLOUT   DD SYSOUT=A,DCB=(LRECL=121,BLKSIZE=121,RECFM=FBM)
//SYSLIN    DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//SYSPRINT DD SYSOUT=A,DCB=(LRECL=125,BLKSIZE=0129,RECFM=VBA)
//WHIMCG PEND
```

Figure 11: Procedure for Executing WHIMSI

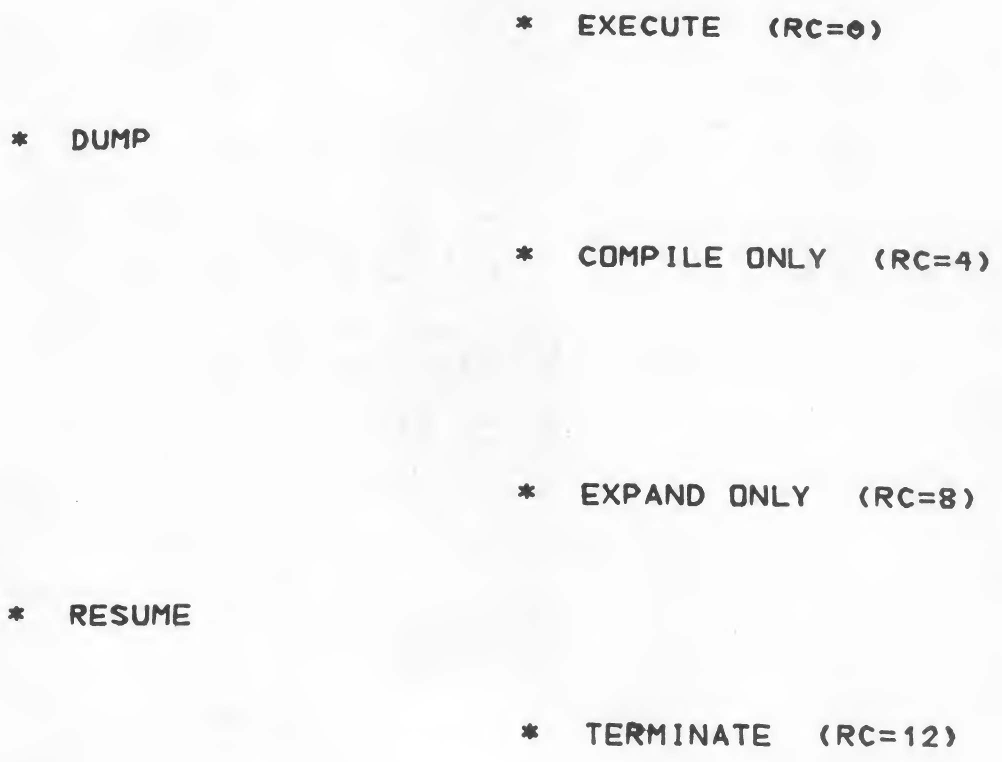


Figure 12: EXIT Menu

The procedure provided must immediately follow the JOB card (and any ROUTE and JOBPARM cards) in the job stream. It may subsequently be executed as any other procedure. The Unit, Vol and Dsn parameters are set with defaults, but may be overridden to specify unit, volume and dataset information (respectively) for the program file to be edited. This file must have been previously created and initialized by program FINIT (See below) or an error will occur. The Name parameter may be used to specify the name of the external procedure for the PL/1 program to be created. If it is omitted, the name will default to TEST.

File_Maintenance_Programs. Two additional programs are provided to complement the use of the main system. These programs are FINIT and WHIMOVE, and are used to initialize and move program files, respectively. The requisite JCL packages for both programs are presented in Figure 13.

In initializing a program file, it is important to specify one parameter in addition to those generally required of disk datasets. This parameter (Mapsize) is specified, as shown, in the PARM field of the EXEC statement. It specifies the size, in bytes of the file map (See Appendix III) to be created. The value specified here is adequate for small programs, but should be increased for larger ones. It is important to specify adequate space, since it cannot be increased without re-initializing the

```
//INIT   EXEC  PGM=FINIT,PARM='MAPSIZE=2000'
//STEPLIB   DD UNIT=3330,VOL=SER=SYS301,DISP=SHR,
               DSN=WHIMSI.LIBRARY
//SYSPRINT DD SYSOUT=A,DCB=(RECFM=VBA,LRECL=125,BLKSIZE=129)
//WHIMSI    DD UNIT=3330,VOL=SER=SYS301,DISP=(,KEEP),
//             SPACE=(TRK,2),DSN=PROGRAM
//             DCB=(DSORG=DA,RECFM=F,BLKSIZE=1000)
```

```
//MOVE    EXEC PGM=WHIMOVE
//STEPLIB DD   UNIT=3330,VOL=SER=SYS301,DISP=SHR,
//             DSN=WHIMSI.LIBRARY
//WHIMSI  DD   UNIT=SYSDA,VOL=SER=TEMP,DISP=(,KEEP),
//             DSN=PROGRAM,DCB=(DSORG=DA,RECFM=F)
//             SPACE=(TRK,2)
//INPUT   DD   UNIT=3330,VOL=SER=SYS301,DISP=OLD,
//             DSN=PROGRAM
```

Figure 13: JCL for Utility Programs

file.

Upon initialization, the file will be filled to the limit specified by the primary allocation. No secondary allocation will be used. That portion of the file remaining after space for the map has been allocated will be used to store any strings entered by the user. If more space becomes necessary for these strings, program WHIMOVE may be used.

Program WHIMOVE moves a WHIMSI program file from one disk dataset to another. The output file is specified by the WHIMSI DD card. The space allocated for the new file must not be less than that allocated to the input file, or a diagnostic will be generated and the program abnormally terminated. However, if the newly allocated space is greater, it will be used and a file may thus be expanded. Note that, if a different device is used for output than for input, the space must be counted in terms of data blocks rather than tracks. The size of a data block remains unchanged during the move since DCB information is automatically copied.

APPENDIX II:

PROGRAM LOGIC AND ORGANIZATION

WHIMSI consists of 46 control sections with an additional 26 entry points, totaling 55,288 bytes of load module. The source code consists of approximately 6500 lines of Assembler code. Consequently, no attempt will be made to reproduce it here or to fully explicate the program logic. Instead, a general overview is given, which should adequately supplement the reading of the program and its many internal comments.

Initial entry to the program is at entry point WHIMSI in the control section of the same name. This routine produces the HELP menu and thus includes entry point HELP as well. Before entering the HELP routine the first time, WHIMSI allocates the save area stacks and initializes the global variables. It then logs the message 'WHIMSI is up' and enters the HELP mode. If EXIT is selected a call is made to subroutine EXIT to obtain the termination option. If a dump is requested, it is immediately given. Otherwise, control returns to HELP which analyzes the return code. If the return code is negative, Resume was selected, otherwise it is the return code for the program as well.

Selection of either file causes the HELP routine to load the program status for that file's handler. The first time each file is used, the environment is as initialized at the beginning of execution. Otherwise it is as stored when HELP was selected. Upon restoring the status, which includes the program counter, control is passed to the point

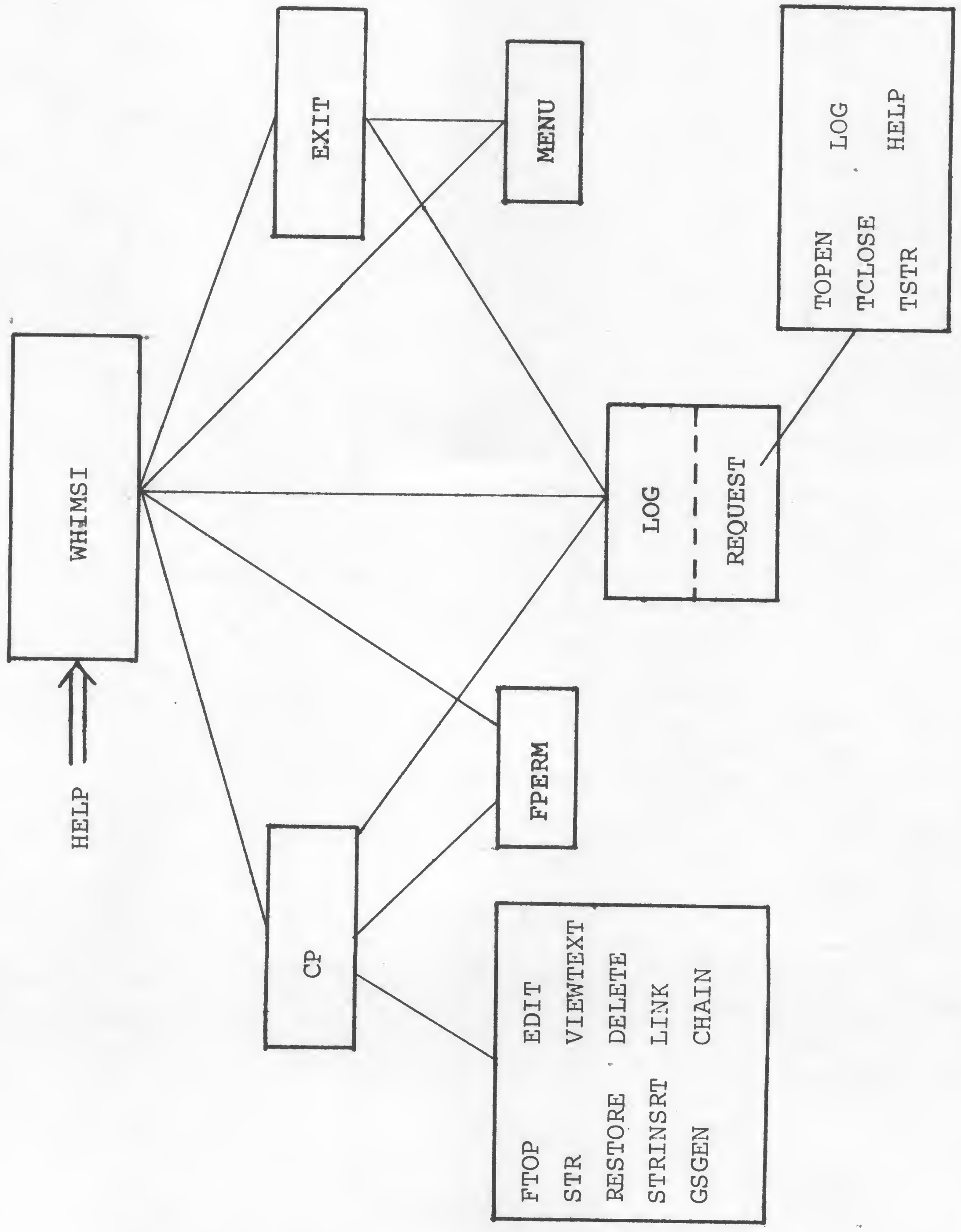


Figure 14: General Program Structure

of interruption.

Processing of each file begins at subroutine CP, the command processor. Although a number of subroutines are used, CP performs the basic analysis of the verbs presented by the user. The majority of the routine is a loop which begins with a request for a verb. When one is received, the appropriate routine for that verb is executed, which in turn obtains any necessary nouns.

Some functions are executed in-line. For instance, the depth control commands require only that the variable DEP be reset. The Back command results in an exit from CP to its calling routine. Since navigation commands are implemented as recursive calls to CP with the new location, the exit has the effect of popping the stack which records the path. The Top command uses a subroutine to locate the top segment and then proceeds in-line to reset the stack.

Attention Processing. The REQUEST macro generates a call to the subroutine of the same name. It is this routine which issues the DISPLAY macro to produce the image on the screen, which includes the last Query issued by the program. It is also here that the illumination of the programmed function keys is controlled. Upon thus setting the stage, the routine enters a loop which is exited upon receipt of an appropriate attention.

When an attention occurs, the operating system returns control to the routine, which proceeds to analyze it. Upon determining the type of attention, it checks its input parameter list for a corresponding request. A separate request must exist for each key that is enabled, and for each class of noun that may be specified by the light pen. When it is found that the information received was on the request list, the loop is exited and the reply formulated. The only exception is when the HELP key is pressed. In this case, the HELP macro is issued to pass control to the HELP routine. Upon return execution continues as though the key press were invalid. That is, another is obtained.

Although identification of a key press provides all the necessary information to be returned, a pen detect must be further analyzed. The analysis routine searches the display structure for an element whose window and buffer specifications correspond to the screen coordinates and buffer address of the light pen interrupt.

Searching of the linear lists such as a menu list or a set of links is straightforward. However, the tree - structured segment display is searched in a special way to avoid a complete traversal. Since the segment display is nested, any object within it is also within the window of the top segment. In general, any element subordinate to another is contained within its window. Moreover, the windows of all subordinates are non-overlapping and so only one path of the tree need be searched. The search proceeds along a path of narrowing windows -- all containing the point detected -- until an element is found whose corresponding buffer segment corresponds to the buffer address of the interrupt. If the search fails, the object is unidentified and the attention is treated as invalid.

Once the desired information is obtained, it must be coded for return. The input parameter list includes an ordered list of request types. The information returned includes an integer specifying the position in that list of the type of information obtained. This is stored in the high order byte of a register and the remainder of the integer contains the data associated with the graphic element indicated in case of light pen detect, or zero in case of a key press. The integer is used by the command processor to index into the list of processing routines, and the remaining information is used to locate the program structure element indicated.

File Processing. All access to the disk files is mediated by a list of FTDs (See Appendix III) and subroutine FIOS. This subroutine has two entries, FIOSR and FIOSW, for reading and writing respectively. The only parameter for either is a chain of FTDs for the strings to be obtained or written.

FIOS first sorts the string's file addresses into ascending order so that strings within the same block may be processed together, reducing IO activity. The processing of each FTD requires first that the string's address be converted from offset in the file to block number and offset in the block. Since a string may span blocks, a check must be made. If necessary, the request is divided and a dummy FTD is generated for the remainder of the string. The dummy FTD is inserted into the chain in such a manner that the actual chain is not modified.

APPENDIX III:

PRINCIPAL DATA STRUCTURES

WHIMSI uses a number of data structures to control its operation as well as to maintain the hierarchic text created. These structures may be divided roughly into three classes. The first class is also the smallest. It is concerned with coordinating the other structures and maintaining the global environment of the system. The other two classes of structures deal with the file subsystem and the graphics subsystem respectively.

Global Organization. Since the global navigation system allows free movement between two files and the HELP frame with complete status saving, three distinct program environments must be maintained. Furthermore, since two of these environments may exist within the same file - manipulation subsystem, a means is needed to maintain the required structures independently of fixed storage addresses.

All variables which have more than local significance are part of a tree - like structure whose root is maintained in general register 13. Consequently, all that is needed to restore a program's environment is the contents of that register. In compliance with OS standards, R13 is also used to point to the current register save area. Thus the save area is considered as part of the overall structure. Normal conventions are followed in doubly linking save areas, but the first word of each save area (not normally used) contains a pointer to a global vector which ties together the remainder of the current program environment.

The program environment consists of a save - area stack, the file control structure, and the graphics control structure. The stack is controlled by the first three words of the global vector. Word four points to the Graphics Control Vector (GCV), and word five points to the File Control Vector (FCV).

Each of the three program environments possesses a region in core (in Subpool 3) which is used as a contiguous stack. The first word of the global vector points to the next available save area in the stack. The second word indicates the stack overflow address, and the third word points to the current save area. In spite of this arrangement, the normal double linkage is maintained between save areas, since some local variables are also allocated from the stack in order to maintain reentrant status.

<u>Subpool Allocation Conventions.</u> The remaining two structures are allocated within separate subpools for programming convenience. From time to time, one or another of these structures will be reinitialized by freeing its respective subpool. Although the GCV and FCV are permanent and are embedded in the program, the remaining elements of the respective structures are obtained by GETMAIN and are thus dynamic. The graphics structures for the Program file,

the Manual and HELP are allocated from subpools 4, 5 and 6 respectively. File control structures are allocated from subpools 8 and 9 for the Program and Manual files respectively. The HELP routine uses no files.

The Graphics Structure

The Graphics Control Vector (GCV) consists of six words, of which five are used. The second word is reserved for a feature not implemented.

Word one points to a Query item if one exists; otherwise it contains zero. A Query item is generated as a string at the top line of the display, and is used to convey requests or questions to the user. It consists of a character string, preceded by a halfword specifying the length of the message in bytes.

Word three of the GCV points to the root of a tree - like structure which represents the segment portion of the display when it exists. Otherwise, the structure is a string element or a menu. These items are discussed more fully below.

Word four points to a list of elements representing the links to be displayed. These elements have a format similar to that of the segment display elements, and are discussed in the same context below.

Word five maintains a "list" of structurable (Undefined) segments in the display. Since only the first element of that "list" is ever needed however, only that first element is included. The low-order three bytes of the word contain the address of the core image of a structurable segment if one is displayed. The high-order byte indicates the number of structurable segments in the display. This information is used by the command processor to help interpret structuring commands and to determine whether such commands should be enabled.

Word six -- the last word of the GCV -- contains the subpool number of the graphics structure in its high-order byte. The remainder of the word is zero and thus the word may be used for subpool specification by the GETMAIN and FREEMAIN macros. This value is also used for allocation of file control structures, since the appropriate subpool number is always exactly four greater than for the corresponding graphics structures.

Segment Display Elements. As mentioned above, a special structure is maintained to represent the current segment, string or menu display. Although each element of this structure varies according to the type of object displayed, all share a common prefix. This prefix is also shared by the link display element which is in a separate list.

The common prefix consists of six words. The first word contains a type indicator in its high-order byte, and a data element in the remaining three bytes. The next four words specify the bounds of that portion of the screen within which the object will be displayed, and the final word specifies the location of the corresponding graphic orders in the 2250 display buffer.

Nine different type codes are accepted as valid. String, menu and link elements are specified by "S," "M" and "L" respectively. The remaining six types correspond to segment types. CAT, Repeat, IF, Text, Null and Undefined segments are specified by their initial letters. This code indicates to the program which type - dependent appendage (if any) exists with the element.

The data item is generally a pointer to the core image of the object represented. For instance, a string element points to the corresponding File Text Descriptor (FTD), and link and segment elements point to their respective objects in the file map. However, a menu item does not refer to an object. Instead its data item is an integer indicating which item of the menu it represents. This integer is used by the command processor to index into a list of service routines which are thus selected by the user.

The display limits (or window) are specified by four floating point numbers between zero and one. They represent the minimum X and Y and the maximum X and Y coordinates, respectively. Coordinates are normalized such that zeroes represent the lower left corner and ones represent the upper right corner of the screen. All graphics routines (except the buffer generation routine) work in terms of these normalized units, thus maintaining a degree of device independence.

The buffer location of the corresponding graphic orders is indicated by two halfwords. The first specifies the buffer address of the first graphic order, and the second is the first address beyond the last order. These addresses thus delimit the object and, together with the window information, serve to help identify the object of a pen detect.

Any subsequent words of the display structure element depend on the type of element. String and Null segment elements have no such appendage. Since the menu elements form a list, the first word of their appendage is a pointer to the next element of the list, or zero. The remainder is a halfword giving the length of the menu item's string, followed by that string. Link elements are also chained and so the first word of the appendage points to the next link element. A subsequent word points to the FTD for the link's name. Elements for segments (other than Null) have a pointer to a string element for the segment's description, followed by pointers to elements for any subordinate segments, and a list of coordinate pairs describing the box which is to represent the segment in the display.

THE FILE CONTROL STRUCTURE

The File Control Vector (FCV) consists of seven words. The first two words contain the DDname for the related file and are used only for reference in reading dumps. The remaining words contain pointers to the DCB, the core image of the file map, the file buffer, the current data block, the first free flag and the first FTD, respectively. The low order three bytes of the data block field contain the relative block number of the record currently in the buffer. The high order byte is set to all ones to indicate the presence of updates in the buffer; otherwise it is zero.

The File Map. The structure of the program file (and thus of the program) is maintained in a file map. A core image of this map is maintained and is copied to disk whenever the dialog indicates that changes are to be made permanent. Reversal of changes is made by simply refreshing this map from disk. All pointers contained within this map are byte offsets from the beginning of the file. Locations within the map are represented by halfword fields, and string locations (outside the map) are addressed by the low-order three bytes of a full word. Translation of byte offsets to actual disk addresses is performed by a special file control module (See Appendix II).

All objects within the map are dynamically allocated, with the exception of a static header which is used to locate the remaining elements. This header consists of 11 words subdivided into 6 fields. The first field is a File Storage Management Block such as is described below. It forms the first element of a chain which controls storage outside the map. Storage within the map is controlled by a chain of File Map Storage Management Blocks, the first of which is at offset 12 from the beginning of the map. At offset 18 there is a pointer to the top (root) segment of the program followed by a halfword indicating the number of bytes in the map. The halfword at offset 22 is unused and is followed by the first floating link at offset 24. This floating link constitutes the first element of the chain of Undefined segments.

The File Storage Management Blocks (FSMB) form a doubly linked chain of free areas in the file outside the map. Each FSMB consists of pointers to the next FSMB, the previous FSMB, and the free area. The final fullword indicates the length in bytes of the free area. These elements are sorted so that the corresponding free areas have successively higher addresses in the file. Each FSMB requires 12 bytes and is allocated within the file map.

The File Map Storage Management Block (FMSMB) performs the analogous function with respect to the file map storage. These elements are doubly linked and sorted as are the

FSMBs, but their format is different. Since they must be allocated within the map, each exists within the first six bytes of a free area. It consists of a pointer to the next FMSMB followed by a halfword indicating the length of the free area and a pointer to the previous FMSMB. Since this element must be embedded within the free area, map allocation is required to be in multiples of eight bytes.

Each segment in the map -- regardless of its type -- occupies 20 bytes. The first byte is a character indicating the type of segment, according to the same convention described above for display elements. The next nine bytes constitute a string element (as described below) for the segment's description. The remaining fields contain pointers to the superordinate segment, the first link originating from the segment (or the first floating link if there is none), and up to three subordinate strings or segments. The sequence of subordinates is as specified in Part III under the heading of Fantic Space. Note that when there are less than three subordinates, the remaining fields are set to zero.

A link also occupies 20 bytes of the map. Its first byte is "L" to identify it as a link, and the string element for its name follows. Since a link may be associated both with a segment and with a chain, it is part of two doubly linked lists. Following the pointer to the segment it references are thus two pairs of pointers. The first pointer of each pair is a forward pointer and the second is a back pointer for the respective list. The first list contains all links originating at a given segment, or all floating links. In the case of floating links, the back pointer of the first link of the list is undefined and the forward pointer of the last is zero. In the case of links from a segment, the first points back to the segment, and the last points forward to the first floating link. The second pair of pointers connects the list of links in a chain or otherwise contains zeroes.

The string element mentioned above consists of nine bytes and is generally embedded in another element. However, it may stand alone if it is a repetition control clause, Boolean expression or Text string. In any of these latter cases, it is preceded by an "S" to identify it as a string element. The first three bytes are the low-order bytes of a word, of which the high order byte contains a type code. These bytes contain the offset in the file to the referenced string, which is stored outside the map. The remaining three halfwords specify the current length of the string, the number of bytes currently allocated to it (not less than the length), and the allocation limit (not less than the allocation). The allocation limit is set upon creation of the string and serves to prevent excessive growth of a string. In general, this limit is not set lower than will ever be required for proper operation of the program. Thus it serves mostly as an internal program check.

<u>Free Flags.</u> In order to help maintain the integrity of the file structure in case of sudden (abnormal) termination of the program, a special allocation convention is followed for file space outside the map. Upon removal of such space from the structure specified by the map, the space is added to a Free queue instead of being made available for reallocation. When the map is subsequently moved to disk, this space is then freed. Thus if changes to the map are prevented by some event from moving to the disk, the structure remains intact as it was before the changes were made. This applies to abnormal terminations as well as to use of the Restore command.

Free flags form a linked list headed in the FCV. Each element of this list occupies ten bytes. It consists of a pointer to the next Free flag, followed by the offset in the file to the area to be freed and a halfword giving the length of that area.

<u>File Text Descriptors.</u> In order to facilitate manipulation of a string in core, a File Text Descriptor (FTD) is created for each. It consists of all the information contained in the corresponding string element of the map, together with information used only for the

manipulations. It is the standard object referred to for manipulation of strings, including their movement to and from disk.

Since the FTDs for strings in core form a chain, the first word is a pointer to the next FTD. The second word contains an "S" in the high order byte to identify the FTD, and the low-order three bytes contain the core address of the string. The next two halfwords contain the location of the string element in the map and the string's length, respectively. The location of the string in the file follows, followed in turn by halfwords indicating the allocated space and the allocation limit, respectively. The final word of the FTD is used during editing to indicate the position of the cursor (if any) in the displayed string, or zero if there is none.

APPENDIX IV:

ASSEMBLER MACROS USED

In addition to the macros supplied with the operating system, WHIMSI uses 103 macros which were created to assist in generating repetitious code or simply to help control the use of data structures.

The operating system macros used provide services in four major categories. The services of GPS Basic attention handling are used to obtain information from the user. IO operations to and from the 2250 terminal (other than attention handling) are performed by EXCP and related macros. Those system macros associated with BDAM IO are used to process the disk files, and QSAM is used for the WHIMSI log. Finally, the system's storage allocation macros are used to maintain the dynamic data structures.

Those macros created for use in WHIMSI may be considered in five categories, and are stored in as many separate libraries. The MACLIB group consists of general - purpose programming macros, most of which are used to help structure the program. The CPLIB group contains those macros used exclusively by the command processor and some of the utility programs. FLIB contains macros used for creating and manipulating elements of the file structures. Display structure processing is aided by macros from the DSLIB group. TERMLIB macros are used to perform actions directly involving the terminal, including some which provide a degree of independence from the terminal type.

The descriptions of these macros included here are incomplete for considerations of space. More complete descriptions, and examples of their use, may be obtained from the program listings.

In reading the macro definitions, it is useful to keep in mind some conventions used in their design. In most cases, the macro is used to modify or set the value of a single parameter. When this is the case, that parameter is coded first. It is generally assumed that this parameter specifies a register. If the value is a single word, it is placed in the register. Otherwise, the register is used to address the longer object in core. Except where only registers are allowed as parameters, they must be enclosed in parentheses. Literal strings are always enclosed in quotes (').

MACLIB MACROS

In addition to the program structuring macros in this library, there are three which perform functions needed throughout the program.

From time to time, it becomes necessary to move a string whose length is not known at compile time and thus must be computed. Furthermore, such a string's length may exceed the maximum of 256 allowed by the machine's MVC instruction. The MVL macro provides the needed service by allowing the length to be unlimited, and specified by a register.

Another function frequently needed is the location of a character within a string. The INDEX macro provides this service, returning the location to a register. A zero is returned if the character is not found.

The ASSERT macro is used to help locate errors during debugging of the program. It has one mandatory parameter which specifies the state of the condition code. This state is coded as in the IF macro described below, and is used to indicate a condition which must hold if the program is correct. If the condition does not hold, an Abend is generated producing a dump to show just what went wrong. The second parameter is an optional character string which will be printed on the program output prior to the Abend.

Use of this macro has proved highly successful in detecting program bugs since it prevents errors from compounding before the Abend occurs. Indeed, during the debugging process, most Abends occurred as a result of this macro.

<u>Subroutine Linkage Macros.</u> A procedure may be delimited by the PROC and PEND macros. PROC takes a name which is the object of the PERFORM macro. Since the name generated by PROC is external, the procedure may be entered from any control section. It differs from the closed subroutines in that no saving of registers takes place, and no parameters are passed. Instead, the performed procedure executes as if it were in-line code, except that registers 14 through 1 might be altered.

Closed subroutines begin with the WENTER macro. Since the convention is followed that each subroutine is a complete control section, the END statement terminates the subroutine. Exit is by means of the WEXIT macro. WENTER performs the necessary register saving and allocates a new save area from the global stack, together with any local variables which must be allocated. The option exists to copy register one into another register and to apply a symbolic name to it for later use. If the option is not selected, the parameter list is lost. CALL is used to enter a subroutine. This macro functions similarly to that provided by the system, except that registers may be

The SELECT macro simulates a <u>case</u> statement. Its two parameters specify the number of segments to be selected from, and the one to be selected, respectively. The first parameter should be a literal, and is used in conjunction with ENDSEG to determine which is the last of the segments coded. Thus each segment associated with this macro should be concluded with ENDSEG. The second parameter may be the name of a fullword or a parenthesized register specification. In either case the variable addressed must contain an integer greater than zero but not greater than the number of segments available. Execution proceeds to the selected segment, bypassing the remainder. If the second variable does not correctly specify a segment, an abend is generated. However, a keyword parameter "ERR=" may be used to select other action. The available options are SKIP and IGNORE. If the former is selected, a range error simply causes all segments to be bypassed. In the latter case, the code generated assumes that the values will always be correct and thus the behavior in case of error cannot be predicted.

Repetition is provided by the REPEAT macro, which causes the segment it begins to be repeated as a loop. Exit from the loop is obtained by use of the EXITIF macro. This macro has one parameter and is coded exactly as in the IF macro. If the condition is satisfied, execution proceeds to the statement following the repeated segment. Otherwise, repetition continues.

The segments created by these macros may be nested to a depth of 32. Although such a depth is not likely to be realized, nesting does become somewhat deep in some cases. Consequently, some facility is needed to assure that no error in nesting occurs. This is provided by the LEVEL macro. At any point in the program, the programmer may code the LEVEL macro with a literal integer as its parameter. If this integer equals the depth of nesting at that point, no action is taken and no code is generated. However, if the value is in error, a diagnostic is generated.

CPLIB MACROS

Of the macros in this library, HELP and LOG are the most important. HELP is used to enter the HELP frame, and LOG is used to make an entry in the program log.

HELP first logs the fact that it was used, and proceeds to provide linkage into the HELP routine. Since the routine which issues HELP is actually a subroutine of the HELP routine (See Appendix II) special linkage is necessary to maintain the proper relationships. Register 14 is loaded with the address of the HELP routine, and register 15 receives the address at which execution is to resume on return.

LOG generates a call to the LOG subroutine, with a special parameter list providing the message. The message may be either a literal enclosed in single quotes (') or a parenthesized register specification providing a pointer to the message in core. The message in core must be 61 bytes long. If a literal is provided, it will be padded or truncated as necessary. The effect of the macro is that the message will be placed on the LOG dataset, preceded by the Julian date and the time.

DSLIB MACROS

Of the macros used for display structure manipulation, three are of particular interest.

GETGCV is used with a single parameter which is a register specification. The register receives the address of the current Graphics Control Vector (GCV). This macro is embedded within a number of other macros which require access to the GCV.

MENU is used to add an item to the menu list in the display structure. Since the menu is mutually exclusive of any other display, it must not be used when segments or links are to appear. The parameter list specifies the vertical positioning of the item, the data item (or sequence number) to be associated with it, and the string to be displayed. A column number (1 or 2) may be specified. If 1 is used, the string begins at the left of the screen. If 2 is specified, the strings begins in the middle of the screen.

QUERY is used to pass a question or a request to the user. Its single parameter is a string which is to be displayed at the top of the screen.

Each of these macros modifies the display structure only. They have no immediate effect on the display. However, when the DISPLAY macro of the TERMLIB group is issued, the structure is converted to an actual display on the 2250 screen.

TERMLIB MACROS

The most interesting macros of this group have to do with direct terminal control, and with some features to limit the impact of changing terminals.

Terminal control is provided by the REQUEST, DISPLAY and IO macros. The REQUEST macro has two parameters. The first is a register which is to receive the requested information, and the second is a list of types of information which will be accepted. The result is a request to the user for information of the appropriate type. Key presses and light pen detects are screened for appropriateness and ignored if not satisfactory. Upon receiving an appropriate response, control is returned to the program. The DISPLAY macro simply causes the current display structure to be converted to an image on the 2250 screen. No parameters are required. The IO macro has one positional parameter, which specifies the address of a channel program for the 2250. This channel program is performed and checked for completeness. If it fails, a message is sent to the computer operator and the operation is retried in 60 seconds. A keyword parameter COUNT allows the residual byte count to be returned in a register. This is helpful for locating the cursor.

Although several subroutines throughout the program must refer to positions within the display screen, they remain independent of the actual size of that screen and of the relative size of the characters displayed. This is due to a set of macros used by these routines, which provide all device dependent information.

The RTU and UTR macros provide conversion from raster to unit and from unit to raster coordinates respectively. The unit coordinates vary from zero to one and are floating point values. The raster coordinates vary according to the device used, and are generally fixed point values. Due to their device - dependent output (raster units) they are used only in the device - dependent subroutines. Their function is to allow other subroutines to deal only in the unit coordinates.

Two special values which depend in part on the type of device used are provided by the SEGTOP and DELFAC macros, each of which generates a floating point constant based on unit coordinates. The first specifies the Y coordinate of the top of the segment display area on the screen. This value varies with type of device because this area is immediately below the top line, which contains the query, and character size may vary between devices. The other value is a constant used by the routine for generating the box for the IF segment display. Since the space available within the triangular portion for the Boolean expression must be computed, a constant is required which depends in part on the sizes of the characters involved.

Another set of macros is used for incrementing and decrementing of coordinates by any given number of text lines or characters. These are used extensively in formatting displays which contain text. The eight macro names derive from a three-part naming convention. The first three characters of the name are INC or DEC depending on whether the coordinate is to be incremented or decremented, respectively. The second part is either omitted or is the letter "L" to indicate large size characters. The final character is either "X" or "Y" to indicate which coordinate is to be affected. The firsparameter is a floating point register containing the coordinate, and the second parameter is a decimal constant specifying the number of spaces to adjust the value. If the second parameter is omitted, 1.0 is assumed. Thus if the statement "INCY 0,1.5" is coded, floating point register 0 will be incremented by the height of one and one half lines of small characters.

BIBLIOGRAPHY

Bohm, Corrado and G. Jacopini; "Flow diagrams, Turing machines and languages with only two formation rules." CACM 9 No. 5 (May 1966), p. 366-371.

Cavouras, John C.; "On the conversion of programs to decision tables: Method and objectives." CACM 17, No. 8 (August 1974), p. 456-462.

Chomsky, Noam; Aspects of the theory of syntax. M.I.T. press, Cambridge, Massachusetts (1965).

Deutsch, L. Peter and Butler W. Lampson; "An online editor." CACM 10, No. 12 (December 1967), p. 793-799.

Dijkstra, Edsger W.; "Go To statement considered harmful." (Letter to the editor) CACM 11 No. 3 (March 1968) p. 147-148.

Dijkstra, Edsger W.; "Notes on structured programming." in Dahl, O.-J., E.W. Dijkstra and C.A.R. Hoare; Structured programming. Academic Press, New York (1972).

Earley, Jay and Paul Caizergues; "A method for incrementally compiling languages with nested statement structure." CACM 15 No. 12 (December 1972) p. 1040-1044.

Elspas, Bernard, Karl N. Levitt, Richard J. Waldinger and Abraham Waksman; "An assessment of techniques for proving program correctness." Computing Surveys, 4, No. 2 (June 1972) p. 97-147.

Engelbart, Douglas C. and William K. English; "A research center for augmenting human intellect." Proc. 1968 FJCC, Vol. 33, Pt. 1, p. 395-410, AFIPS Press, Montvale, New Jersey.

Fabry, R.S.; "Capability - based addressing." CACM 17, No. 7 (July 1974), p. 403-411.

Floyd, Robert W.; "Nondeterministic algorithms." JACM 14, No. 4 (October 1967), p. 636-644.

German, Steven M. and Ben Wegbreit; "A synthesizer of inductive assertions." Proc. 1975 NCC, p. 369-376, AFIPS Press, Montvale, New Jersey.

Gero, John S.; "A system for CAD in architecture." in Vlietstra, J. and Wielinga, R.F. (eds.); _Computer - aided design._ p. 309-326. North - Holland Publishing Company, Amsterdam (1973).

Gries, David; _Compiler construction for digital computers._ John Wiley & Sons, Inc., New York (1971).

Hansen, W.J.; _Creation of hierarchic text with a computer display._ Argonne National Laboratory Report ANL7818, July 1971.

Hansen, W.J.; "User engineering principles for interactive systems." _Proc. 1971 FJCC_, Vol. 39, p. 523-532, AFIPS Press, Montvale, New Jersey.

Hatvany, J.; "The engineer's creative activity in a CAD environment." in Vlietstra, J. and Wielinga, R.F. (eds.); _Computer - aided design_, p. 113-125, North-Holland Publishing Company, Amsterdam (1973).

Hopcroft, John E. and Jeffrey D. Ullman; _Formal languages and their relation to automata._ Addison - Wesley Publishing Company, Reading, Massachusetts (1969).

Knuth, Donale E.; "Structured programming with GOTO statements." _Computing Surveys_, 6, No. 4 (December 1974), p. 261-301.

Leavenworth, Burt M. and Jean E. Sammet; _An overview of nonprocedural languages._ IBM Report RC-4685, January 17, 1974.

Lee, R.C.T., C.L. Chang and R.J. Waldinger; "An improved program - synthesizing algorithm and its correctness." _CACM_ 17 No. 4 (April 1974), p. 211-217.

Martin, James; _Design of man - computer dialogues._ Prentice - Hall, Inc., Englewood Cliffs, N.J. (1973).

Meadow, Charles T.; _Man - machine communication._ Wiley-Interscience, New York (1970).

Mills, Harlan D.; "Syntax - directed documentation for PL360," _CACM_ 13 No. 4 (April 1970), p. 216-222.

Mills, Harlan D.; "The new math of computer programming." _CACM_ 18, No. 1 (January 1975), p. 43-48.

Nelson, Theodor H.; "Getting it out of our system." in Schecter, George (ed.); _Information retrieval: Critical view._ Thompson Books, Washington, DC. (1967).

Nelson, Theodor H.; "A conceptual framework for man - machine everything." _Proc. 1973 NCC_, p. M21-M26, AFIPS Press, Montvale, New Jersey.

Nelson, Theodor H.; _Computer Lib / Dream Machines._ Hugo's Book Service, Chicago (1974).

Newman, William M. and Robert F. Sproull; _Principles of interactive computer graphics._ McGraw - Hill, New York (1973).

Newsted, P.R.; _FORTRAN program comprehension as a function of documentation._ School of Business Administration Report, The University of Wisconsin, Milwaukee, Wisconsin.

Parnas, D.L. and D.P. Siewiorek; "Use of the concept of transparency in the design of hierarchically structured systems." _CACM_ 18, No. 7 (July 1975), p. 401-408.

Quillian, M. Ross; "Semantic memory." in Minsky, Marvin L. (ed.); _Semantic information processing._ M.I.T. Press, Cambridge, Massachusetts (1968).

Ryan, J.L., R.L. Crandall and M.C. Medwedeff; "A conversational system for incremental compilation and execution in a time - sharing environment." _Proc. AFIPS 1966 FJCC_, Vol. 29, p. 1-22, Spartan Books, N.Y.

Sackman, Harold; _Man - computer problem solving: Experimental evaluation of time - sharing and batch processing._ Auerbach Publishers, Princeton, N.J. (1970).

Schaefer, Marvin; _A mathematical theory of global program optimization._ Prentice - Hall, Englewood Cliffs, New Jersey (1973).

Shneiderman, Ben; "Experimental testing in programming languages, stylistic considerations and design techniques." _Proc. 1975 NCC_, p. 653-656, AFIPS Press, Montvale, New Jersey.

Shwayder, Keith; "Extending the information theory approach to converting limited - entry decision tables to computer programs." _CACM_ 17, No. 9 (September 1974), p. 532-537.

Sime, M., T. Green and D. Guest; "Psychological evaluation of two conditional constructions used in computer languages." _International Journal of Man - Machine Studies_, Vol. 15, No. 1 (1973).

Stevens, W.P, G.J. Myers and L.L. Constantine; "Structured design." _IBM Systems Journal_, Vol. 13, No. 2, p. 115-139 (1974).

Sussman, Gerald Jay; _A computational model of skill acquisition._ Massachusetts Institute of Technology Artificial Intelligence Laboratory Report AI TR-297, August 1973.

Van Dam, Andries and David E. Rice; "On-line text editing: A survey." _Computing Surveys_, Vol. 3, No. 3 (September 1971), p. 93-114.

Wegbreit, Ben; "The synthesis of loop predicates." _CACM_ 17, No. 2 (February 1974), p. 102-112.

Weinberg, Gerald M.; _The psychology of computer programming._ Von Nostrand Reinhold Company, New York (1971).

Wirth, Niklaus; "Program development by stepwise refinement." _CACM_ 14, No. 4 (April 1971) p. 221-227.

Zahn, Charles T., Jr.; "Structured control in programming languages." _Proc. 1975 NCC_, p. 293-295, AFIPS Press, Montvale, New Jersey.